4. 다층 신경망 이해

- 행렬 곱 연산을 이해한다.
- 배치경사 하강법을 이해한다.
- 2개의 층을 가진 신경망을 이해한다.
- 다층 신경망의 경사 하강법을 이해한다.
- 미니배치 경사 하강법을 이해한다.
- 다중분류 다층 신경망을 이해한다.
- 케라스를 이용한 다층신경망의 다양한 구현을 이해한다.

4. 다층 신경망 이해

4.1 행렬 곱 연산

4.2 배치경사 하강법 구현

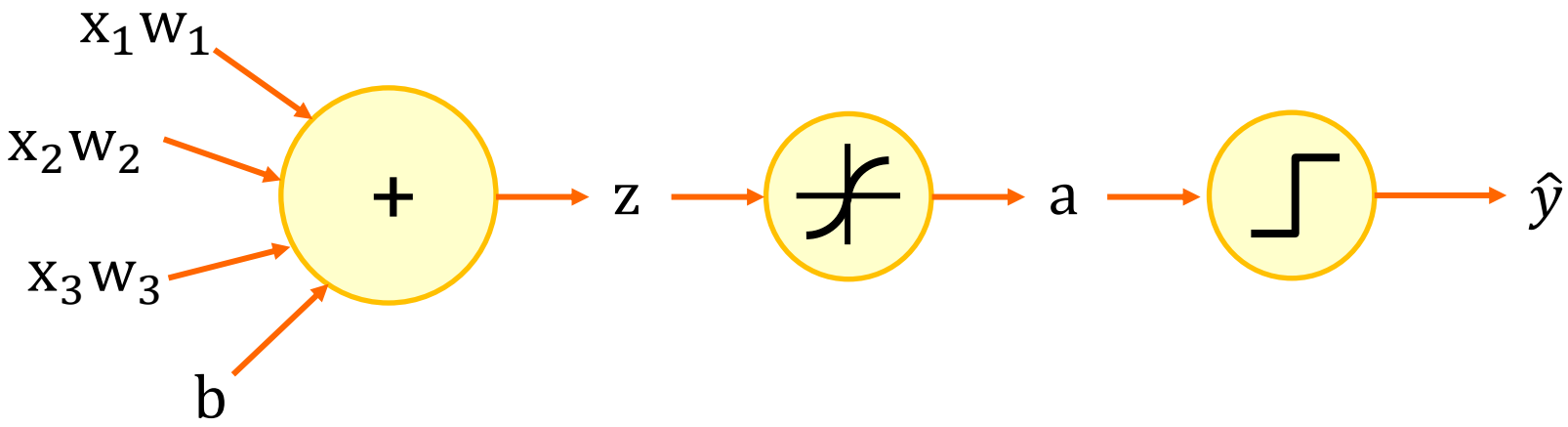
4.3 2개의 층을 가진 신경망 구현

4.4 다층 신경망의 경사 하강법 구현

4.5 미니배치 경사 하강법 구현

4.6 다중분류 다층 신경망을 이해한다.

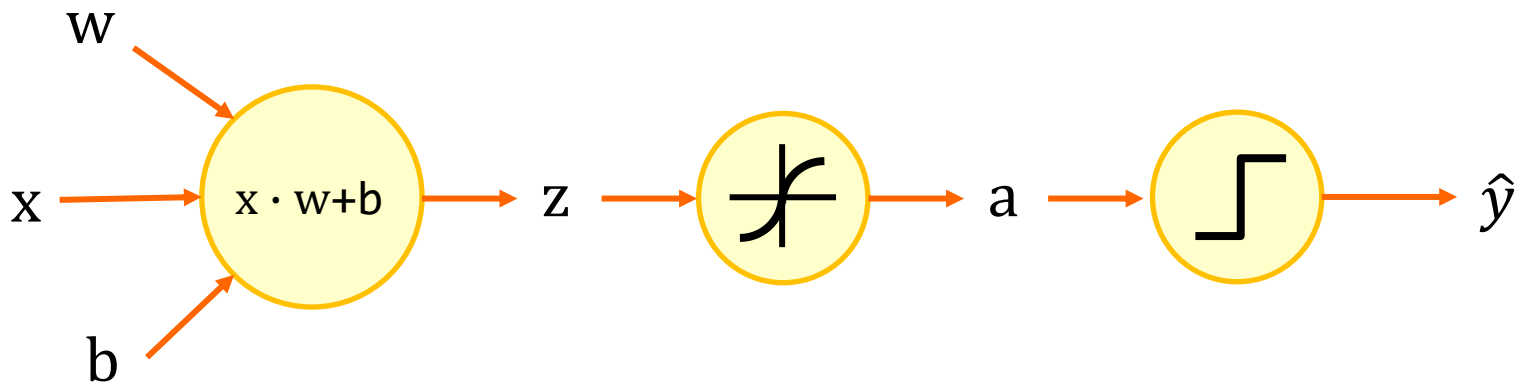
4.7 케라스를 이용한 다층신경망의 다양한 구현




```
def forpass(self, x):
    z = np.sum(x * self.w) + self.b
    return z
```

넘파이의 원소별 곱셈

$$x = [x_1, x_2, \cdots, x_n]$$
$$w = [w_1, w_2, \cdots, w_n]$$
$$x * w = [x_1 * w_1, x_2 * w_2, \cdots, x_n * w_1]$$



점 곱을 행렬 곱셈으로 표현

$$XW = \begin{bmatrix} x_1 & x_2 & x_3 \end{bmatrix} \begin{bmatrix} w_1 \\ w_2 \\ w_3 \end{bmatrix} = x_1 \times w_1 + x_2 \times w_2 + x_3 \times w_3$$

(1, 3)(3, 1) => (1,1)

```
z = np.dot(x , self.w) + self.b
```

4. 다층 신경망 이해

4.1 행렬 곱 연산

4.2 배치경사 하강법 구현

4.3 2개의 층을 가진 신경망 구현

4.4 다층 신경망의 경사 하강법 구현

4.5 미니배치 경사 하강법 구현

4.6 다중분류 다층 신경망을 이해한다.

4.7 케라스를 이용한 다층신경망의 다양한 구현

$$XW = \begin{bmatrix} x_1^{(1)} & x_2^{(1)} & x_3^{(1)} \\ x_1^{(2)} & x_2^{(2)} & x_3^{(2)} \\ & \vdots & \\ x_1^{(m)} & x_2^{(m)} & x_3^{(m)} \end{bmatrix} \begin{bmatrix} w_1 \\ w_2 \\ w_3 \end{bmatrix} = \begin{bmatrix} x_1^{(1)}w_1 + x_2^{(1)}w_2 + x_3^{(1)}w_3 \\ x_1^{(2)}w_1 + x_1^{(2)}w_1 + x_1^{(2)}w_1 \\ \vdots \\ x_1^{(m)}w_1 + x_2^{(m)}w_2 + x_3^{(m)}w_3 \end{bmatrix}$$

$$(m, 3) \cdot (3, 1) = (m, 1)$$

행렬곱 가능과 곱 결과 크기

$$(m, n) \cdot (n, k) = (m, k)$$

첫 번째 행렬의 열(n)과 두 번째 행렬의 행(n)의 크기는 반드시 같아야 한다.

곱 결과의 크기는 첫 번째 행렬의 행(m)과 두 번째 행렬의 열(k)의 크기가 된다.

정방향 계산을 행렬 곱셉으로 표현

$$(364,30) \quad (30,1) \qquad (364,1)$$

$$XW = \begin{bmatrix} x_1^{(1)} & \cdots & x_{30}^{(1)} \\ \vdots & \ddots & \vdots \\ x_1^{(364)} & \cdots & x_{30}^{(364)} \end{bmatrix} \begin{bmatrix} w_1 \\ w_2 \\ \vdots \\ w_{30} \end{bmatrix} + \begin{bmatrix} b \\ b \\ \vdots \\ b \end{bmatrix} = \begin{bmatrix} z^{(1)} \\ z^{(2)} \\ \vdots \\ z^{(364)} \end{bmatrix}$$

$$= \begin{bmatrix} x_1^{(1)} & \cdots & x_{30}^{(1)} \\ \vdots & \ddots & \vdots \\ x_1^{(364)} & \cdots & x_{30}^{(364)} \end{bmatrix} \begin{bmatrix} w_1 \\ w_2 \\ \vdots \\ w_{30} \end{bmatrix} + b = \begin{bmatrix} x_1^{(1)}w_1 + x_2^{(1)}w_2 + \cdots + x_{30}^{(1)}w_{30} + b \\ \vdots \\ x_1^{(364)}w_1 + x_2^{(364)}w_2 + \cdots + x_{30}^{(364)}w_{30} + b \end{bmatrix}$$

그레디언트 계산

$$X^T E = \begin{bmatrix} x_1^{(1)} & \cdots & x_1^{(364)} \\ x_2^{(1)} & \cdots & x_2^{(364)} \\ \vdots & \cdots & \vdots \\ x_{30}^{(1)} & \cdots & x_{30}^{(364)} \end{bmatrix} \begin{bmatrix} e^{(1)} \\ e^{(3)} \\ \vdots \\ e^{(364)} \end{bmatrix} = \begin{bmatrix} g_1 \\ g_2 \\ \vdots \\ g_{30} \end{bmatrix}$$

그레디언트는 X와 E의 행렬곱이다.
그러나 벡터 연산에서 X의 크기는 (364, 30) 이고
E는 (364, 1) 이므로 행렬의 곱을 할 수 없다.

이때 X를 전치하면 행과 열이 바뀌므로
X^T (30, 364) 가 되므로

X^TE 는 (30, 364) · (364, 1) 이므로 곱이 가능하고
결과는 (30, 1) 이 되므로 그레디언트와 같은 행렬을 구할 수 있다.

$$W = W - X^T E$$
$$(364,30)\ (364,1)\ \cancel{XE}$$
$$(30,364)(364,1)$$
$$(30,1)$$

$X \Rightarrow (364, 30)$

$W \Rightarrow (30, 1)$

$\hat{y} \Rightarrow (364, 1)$

$a \Rightarrow (364, 1)$

$err \Rightarrow (a - y) \Rightarrow (364, 1)$

행렬곱의 미분(MatMul)

$$\frac{\partial}{\partial \hat{y}} \frac{1}{2}(\hat{y} - y)^2$$

$$= \hat{y} - y = E$$

$$\frac{\partial}{\partial w} wx + b$$

$$= X$$

$$\frac{\partial J}{\partial \hat{y}} * \frac{\partial \hat{y}}{\partial w} = X^T E$$

Chain Rule 사용

$$\hat{y} = XW + b$$

$$J(w,b) = \frac{1}{2}(\hat{y} - y)^2$$

$$\frac{\partial}{\partial w} J(w,b) = \frac{\partial J}{\partial \hat{y}} * \frac{\partial \hat{y}}{\partial w}$$

전체식을 W로 미분한 경우
사라진 W의 자리에는 뒤에서 날라온
미분값인 E를 쓰고
남아 있는 X는 전치하여
행렬 곱을 한다.
그 결과는 W의 형상과 같은 행렬이 생성된다.

배치 경사 하강법 적용

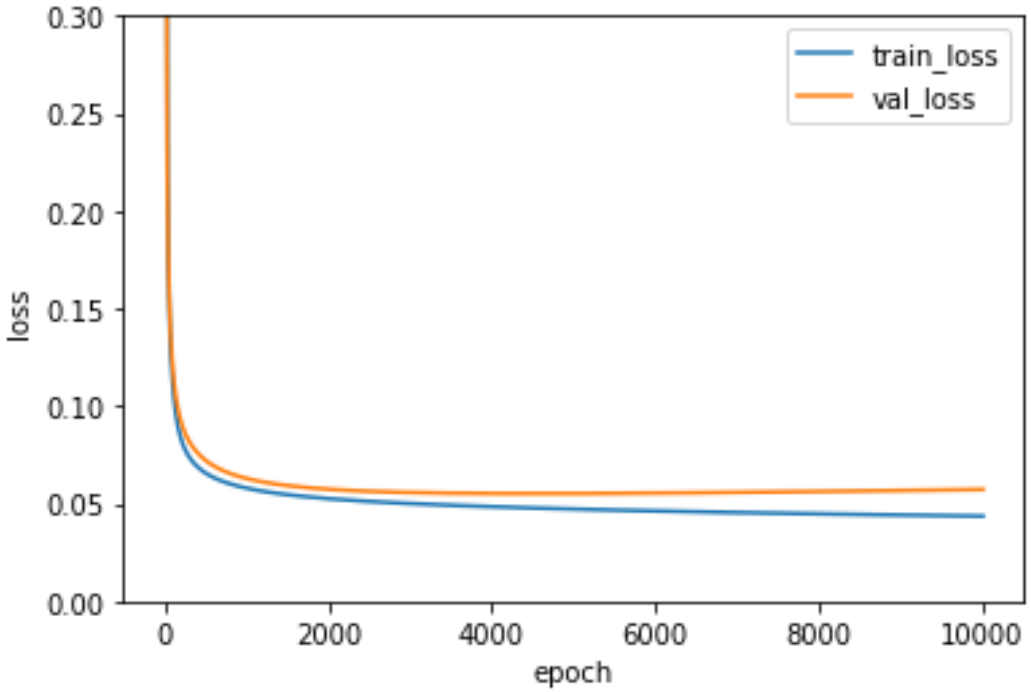
```
def forpass(self, x):
    z = np.dot(x, self.w) + self.b    # 선형 출력을 계산합니다.
    return z

def backprop(self, x, err):
    m = len(x)
    w_grad = np.dot(x.T, err) / m     # 가중치에 대한 그래디언트를 계산합니다.
    b_grad = np.sum(err) / m          # 절편에 대한 그래디언트를 계산합니다.
    return w_grad, b_grad
```

fit () 함수 수정

```
def fit(self, x, y, epochs=100, x_val=None, y_val=None):
        y = y.reshape(-1, 1)                    # 타깃을 열 벡터로 바꿉니다.
        y_val = y_val.reshape(-1, 1)
        m = len(x)                              # 샘플 개수를 저장합니다.
        self.w = np.ones((x.shape[1], 1))       # 가중치를 초기화합니다.
        self.b = 0                              # 절편을 초기화합니다.
        self.w_history.append(self.w.copy())    # 가중치를 기록합니다.
        # epochs만큼 반복합니다.
        for i in range(epochs):
            z = self.forpass(x)                 # 정방향 계산을 수행합니다.
            a = self.activation(z)              # 활성화 함수를 적용합니다.
            err = -(y - a)                      # 오차를 계산합니다.
            w_grad, b_grad = self.backprop(x, err)
            w_grad += (self.l1 * np.sign(self.w) + self.l2 * self.w) / m
            self.w -= self.lr * w_grad
            self.b -= self.lr * b_grad
            self.w_history.append(self.w.copy())
            a = np.clip(a, 1e-10, 1-1e-10)
            loss = np.sum(-(y*np.log(a) + (1-y)*np.log(1-a)))
            self.losses.append((loss + self.reg_loss()) / m)
            self.update_val_loss(x_val, y_val)
```

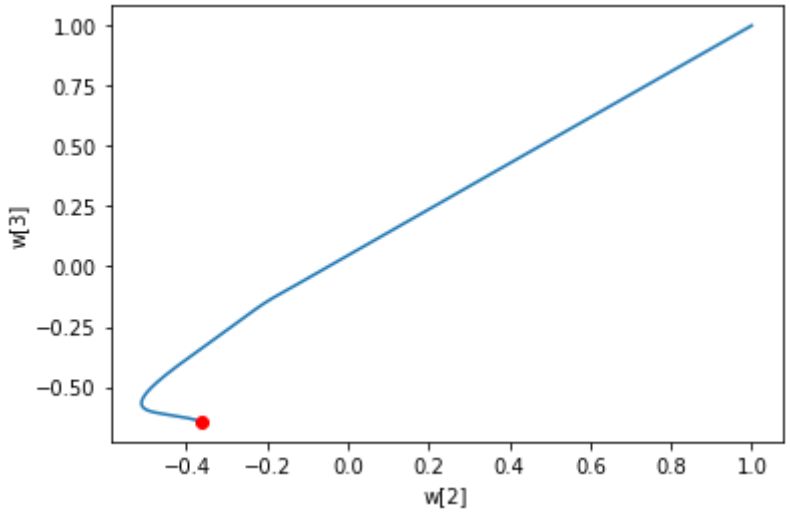
loss율 그래프

```
plt.ylim(0, 0.3)
plt.plot(single_layer.losses)
plt.plot(single_layer.val_losses)
plt.ylabel('loss')
plt.xlabel('epoch')
plt.legend(['train_loss', 'val_loss'])
plt.show()
```



loss율 history 그래프

```
w2 = []
w3 = []
for w in single_layer.w_history:
    w2.append(w[2])
    w3.append(w[3])
plt.plot(w2, w3)
plt.plot(w2[-1], w3[-1], 'ro')
plt.xlabel('w[2]')
plt.ylabel('w[3]')
plt.show()
```



4. 다층 신경망 이해

4.1 행렬 곱 연산

4.2 배치경사 하강법 구현

4.3 2개의 층을 가진 신경망 구현

4.4 다층 신경망의 경사 하강법 구현

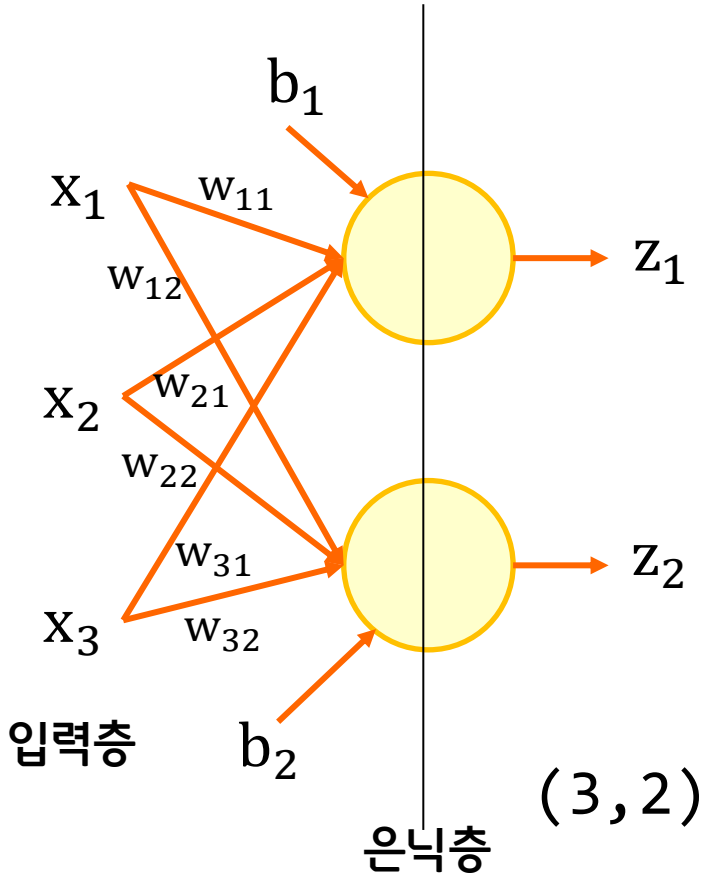
4.5 미니배치 경사 하강법 구현

4.6 다중분류 다층 신경망을 이해한다.

4.7 케라스를 이용한 다층신경망의 다양한 구현

https://playground.tensorflow.org/

하나의 층에 여러개의 뉴런 사용



$$XW_1 + b_1 = z_1$$

$$XW_2 + b_2 = z_2$$

(3,2) => (특성수,뉴런수) => (입력수,출력수)

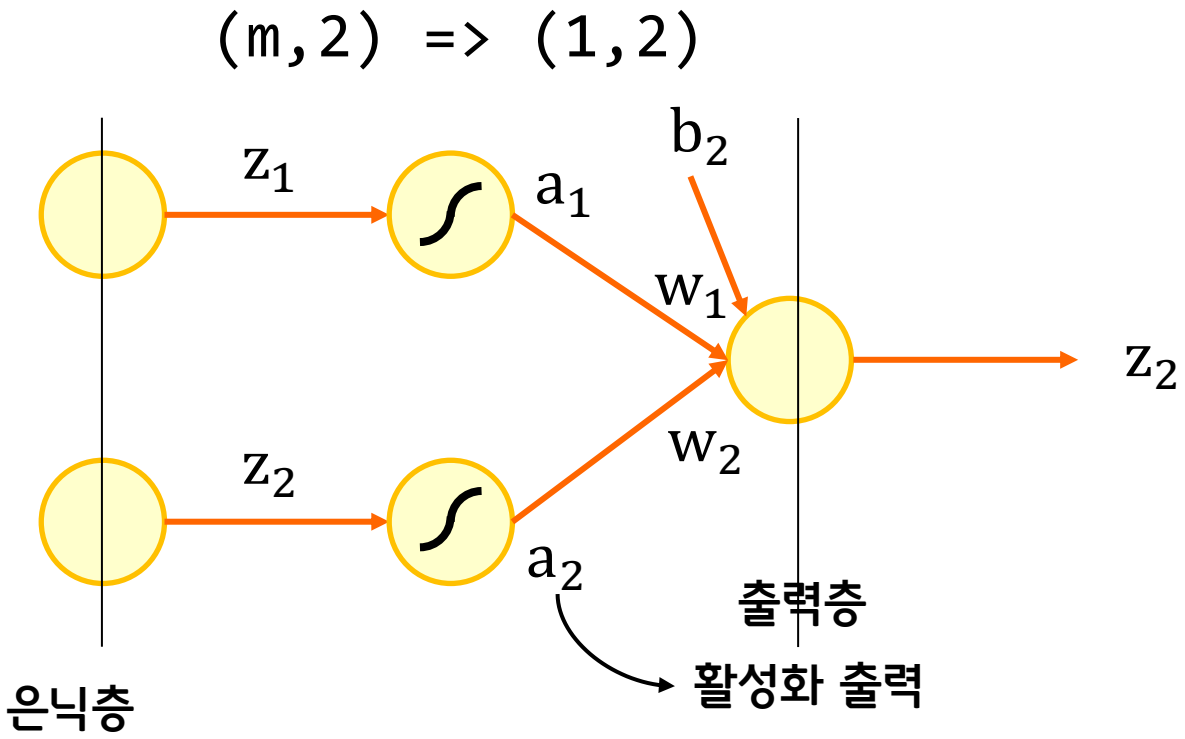
$$x_1w_{11} + x_2w_{21} + x_3w_{31} + b_1 = z_1$$

$$x_1w_{12} + x_2w_{22} + x_3w_{32} + b_2 = z_2$$

$$[x_1 \quad x_2 \quad x_3]\begin{bmatrix} w_{11} & w_{12} \\ w_{21} & w_{22} \\ w_{31} & w_{32} \end{bmatrix} + [b_1 \quad b_2] = [z_1 \quad z_2]$$

(m,n) (n,2) => (m,2)

하나의 층에 여러개의 뉴런 사용



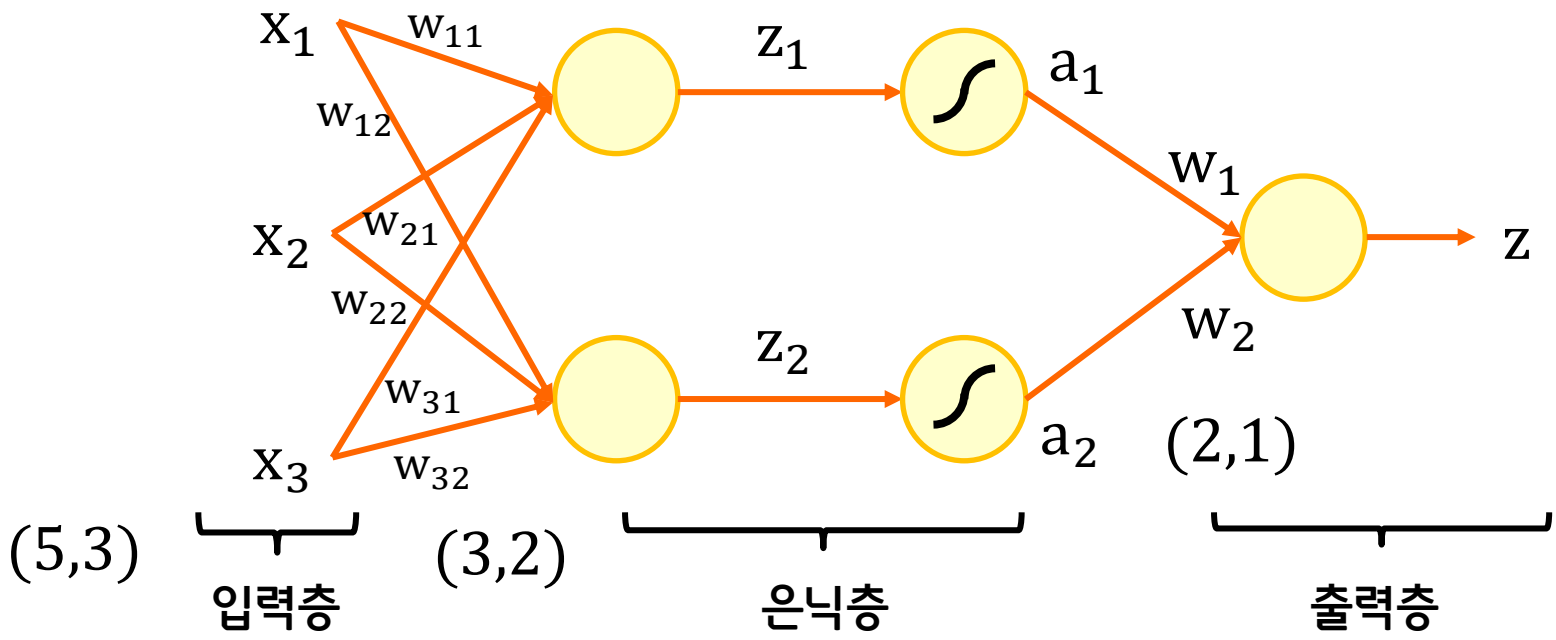
$$a_1 w_1 + a_2 w_2 + b_2 = z$$

$$\begin{bmatrix} a_1 & a_2 \end{bmatrix} \begin{bmatrix} w_1 \\ w_2 \end{bmatrix} + b_2 = z$$

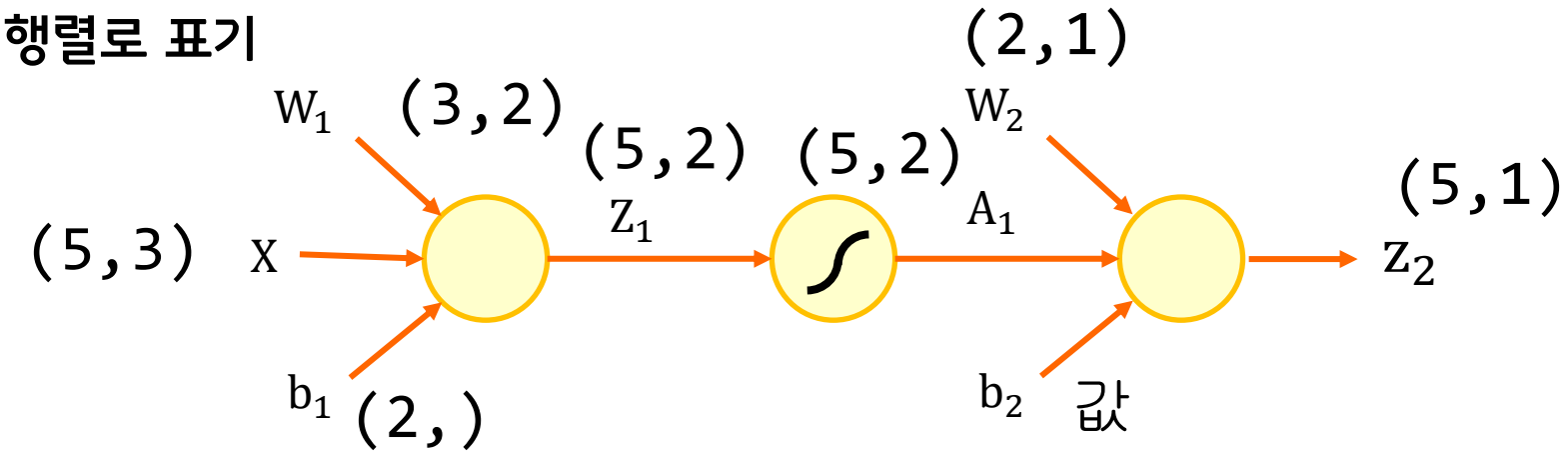
(m,2)(2,1) => (m,1)

$$A_1 W_2 + b_2 = z_2$$

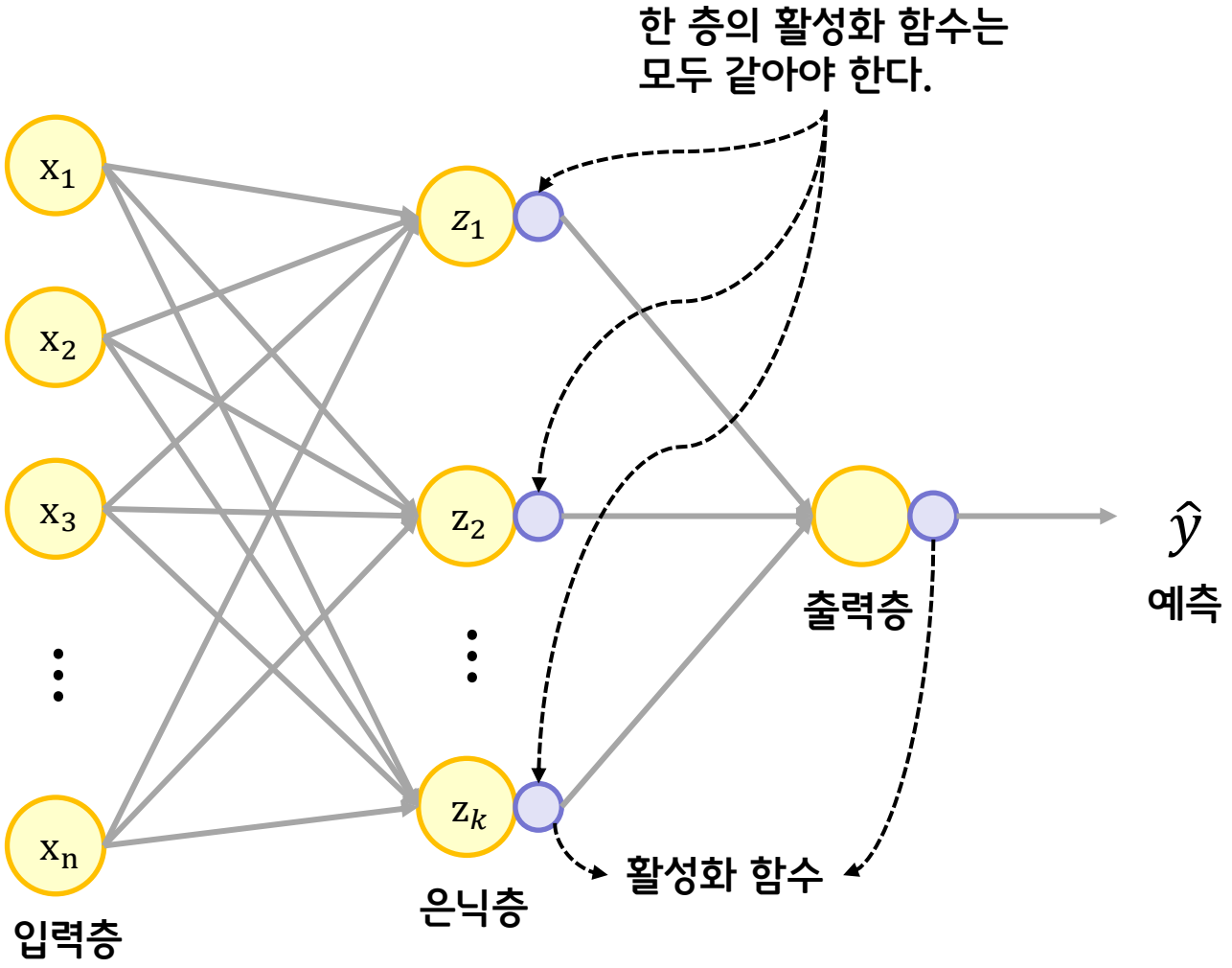
하나의 층에 여러개의 뉴런 사용



행렬로 표기



2층의 완전연결 신경망



X W_1 A W_2

(m, n)(n, k) => (m, k)(k, 1) => (m, 1)

4. 다층 신경망 이해

4.1 행렬 곱 연산

4.2 배치경사 하강법 구현

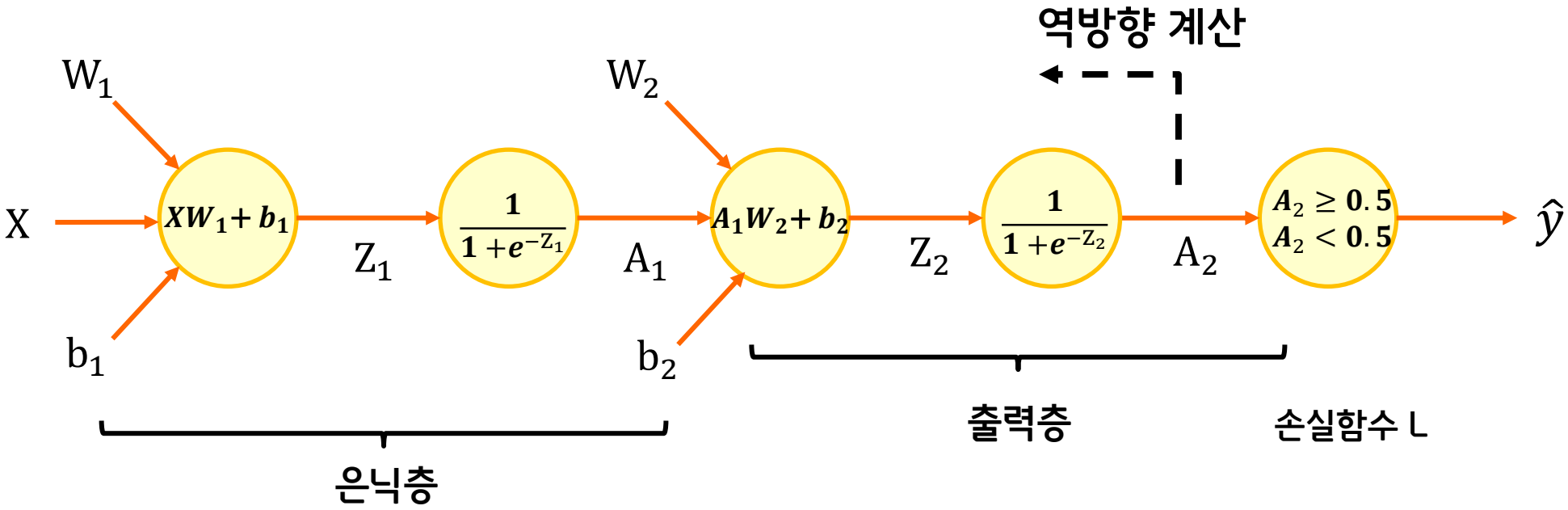
4.3 2개의 층을 가진 신경망 구현

4.4 다층 신경망의 경사 하강법 구현

4.5 미니배치 경사 하강법 구현

4.6 다중분류 다층 신경망을 이해한다.

4.7 케라스를 이용한 다층신경망의 다양한 구현

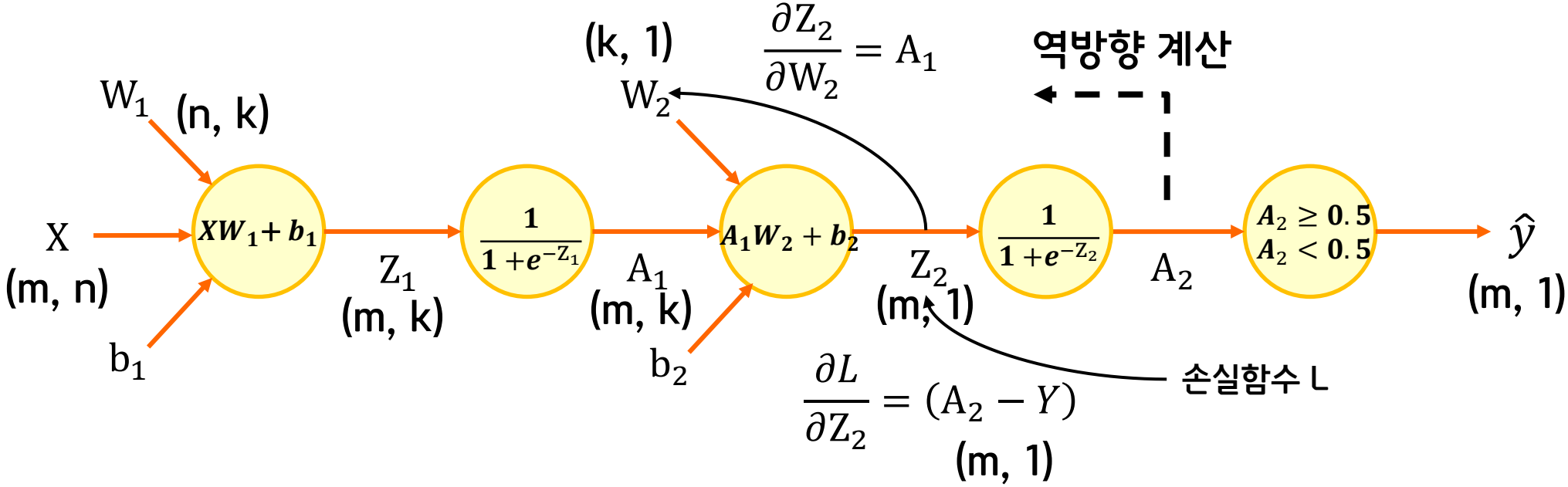

역방향 계산
W1
W2
X
XW1+b1
Z1
1/(1+e^-Z1)
A1
A1W2+b2
Z2
1/(1+e^-Z2)
A2
A2 ≥ 0.5
A2 < 0.5
ŷ
b1
b2
은닉층
출력층
손실함수 L

가중치 W_2 대하여 손실 함수를 미분한다.

$$\frac{\partial L}{\partial W_2} = \frac{\partial L}{\partial Z_2}\frac{\partial Z_2}{\partial W_2}$$

$$\frac{\partial L}{\partial W_2} = (A_1^T \cdot Err)$$

np.dot(self.A1.T, Err)



(k, m) (m, 1) => (k, 1)

A_1^T Err w2_grad

가중치 W_2 대하여 손실 함수를 미분한다.

$$\frac{\partial L}{\partial Z_2} = (A_2 - Y) = \begin{bmatrix} 0.7 \\ 0.3 \\ \vdots \\ 0.6 \end{bmatrix} \} \text{ m개}$$

첫 번째 뉴런의 활성화 출력

$$\frac{\partial Z_2}{\partial W_2} = A_1 = \begin{bmatrix} -1.37 & 0.96 \\ & \vdots \\ 2.10 & -0.17 \end{bmatrix} \} \text{ m개}$$

첫 번째 뉴런의 활성화 출력

가중치 W_2 대하여 손실 함수를 미분한다.

행렬의 구성을 보면 A_1의 크기는 $(\mathrm{m},2)$이고 $(A_2 - Y)$ 의 크기는 $(\mathrm{m},1)$이므로
A_1을 전치하여 $(A_2 - Y)$ 와 곱해야 한다.

그레디언트의 총 합

$$\frac{\partial L}{\partial W_2} = \frac{\partial L}{\partial Z_2}\frac{\partial Z_2}{\partial W_2} = A_1^T(A_2 - Y)) = \begin{bmatrix} -1.37 & \cdots & 2.10 \\ 0.96 & \cdots & -0.17 \end{bmatrix} \begin{bmatrix} 0.7 \\ 0.3 \\ \cdots \\ 0.6 \end{bmatrix} = \begin{bmatrix} -0.12 \\ 0.36 \end{bmatrix}$$

(m, k)(m,1)

(k, m)(m,1) => (k, 1)

m개

타깃과 예측의 차이

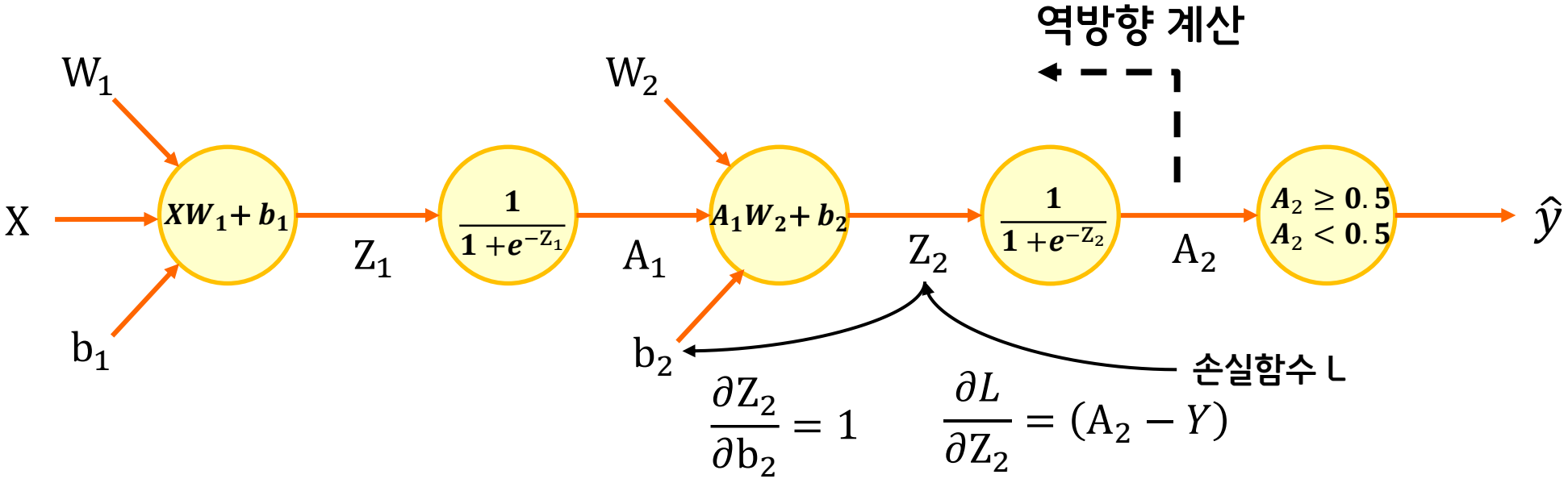
~~$1 \cdot (A_2 - Y)$~~

~~$1 \cdot (m, 1) \Rightarrow (m, 1)$~~

절편 b_2 대하여 손실 함수를 미분한다.

$$\frac{\partial L}{\partial b_2} = \frac{\partial L}{\partial Z_2}\frac{\partial Z_2}{\partial b_2}$$

$$\frac{\partial L}{\partial b_2} = (1^T \cdot Err)$$

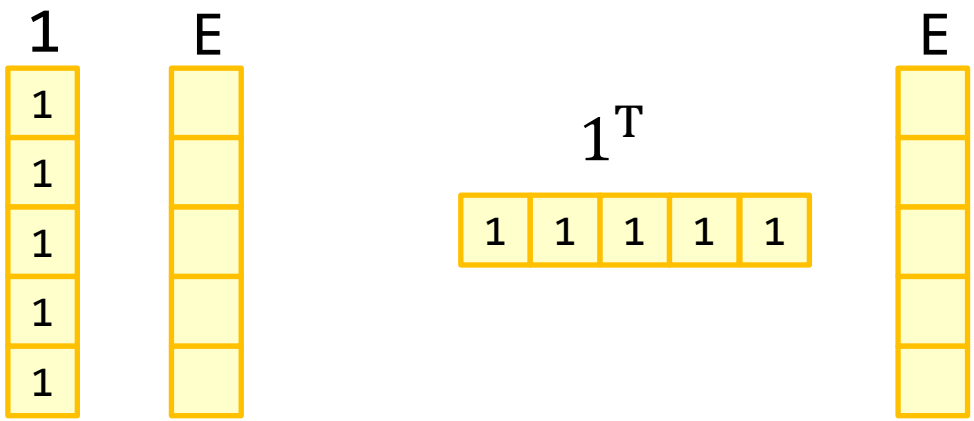


절편 b_2 대하여 손실 함수를 미분한다.

행렬의 구성을 보면 $(A_2 - Y)$ 의 크기는 $(\mathrm{m},1)$이므로 1을 타겟의 개수와 동일하게 벡터로 만들고 벡터를 전치하여 $(A_2 - Y)$ 와 곱해야 한다.

$$\frac{\partial L}{\partial b_2} = \frac{\partial L}{\partial Z_2}\frac{\partial Z_2}{\partial b_2} = 1^{\mathrm{T}}(A_2 - Y))) = \underbrace{[1 \quad \cdots \quad 1]}_{\text{m개}} \cdot \begin{bmatrix} 0.7 \\ 0.3 \\ \cdots \\ 0.6 \end{bmatrix} = 0.18$$

타깃과 예측의 차이




```
b2_grad = np.sum(Err, axis=0)
```

가중치 W_1 대하여 손실 함수를 미분한다.

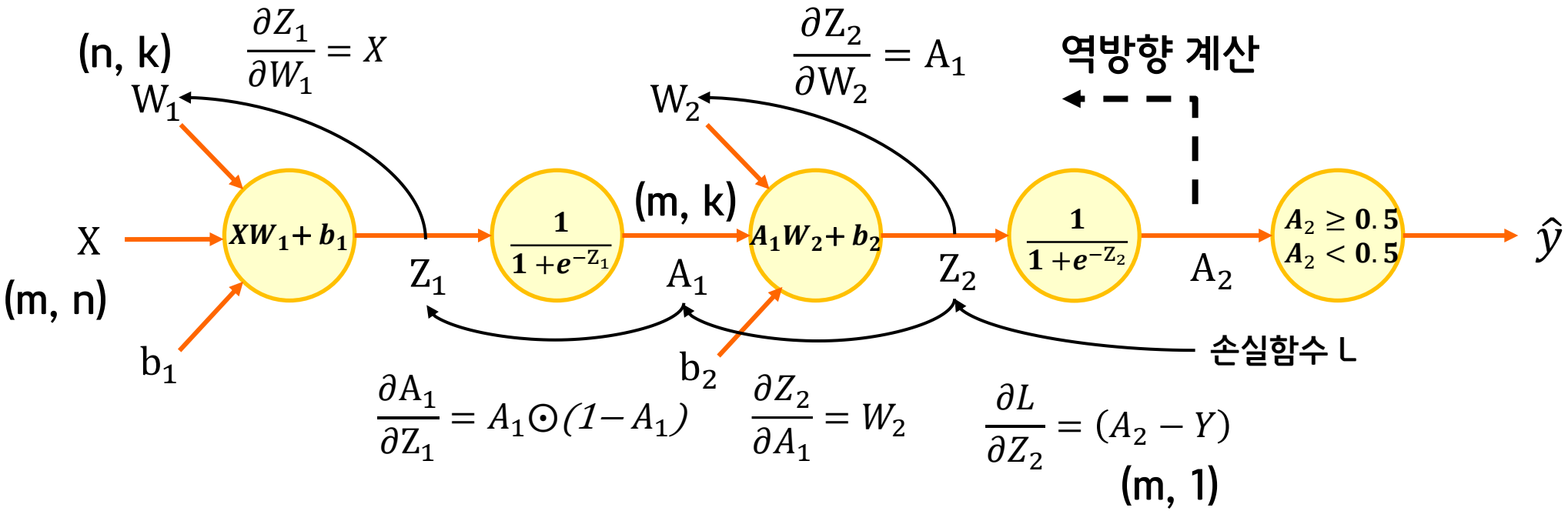
⊙ : 멤버 곱

a ⊙ (1-a)

(m, 1)(1, k) => (m, k)

$$\frac{\partial L}{\partial W_1} = \frac{\partial L}{\partial Z_2}\frac{\partial Z_2}{\partial A_1}\frac{\partial A_1}{\partial Z_1}\frac{\partial Z_1}{\partial W_1} = X^T((A_2 - Y)\,W_2^T \odot A_1 \odot (1 - A_1))$$

(n,m)(m, k) =>(n, k)



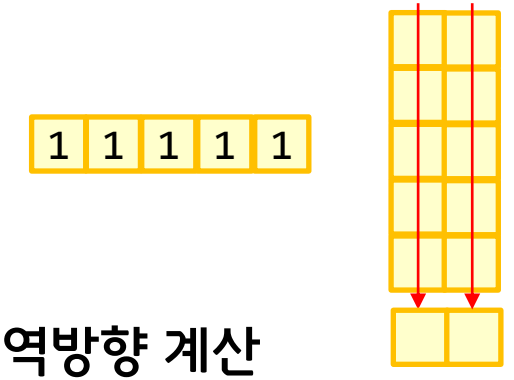
절편 b_1 대하여 손실 함수를 미분한다.

(m, 1)(1, k) => (m, k)

$$\frac{\partial L}{\partial b_1} = \frac{\partial L}{\partial Z_2}\frac{\partial Z_2}{\partial A_1}\frac{\partial A_1}{\partial Z_1}\frac{\partial Z_1}{\partial b_1} = 1^T((A_2 - Y)\,W_2^T \odot A_1 \odot (1 - A_1))$$

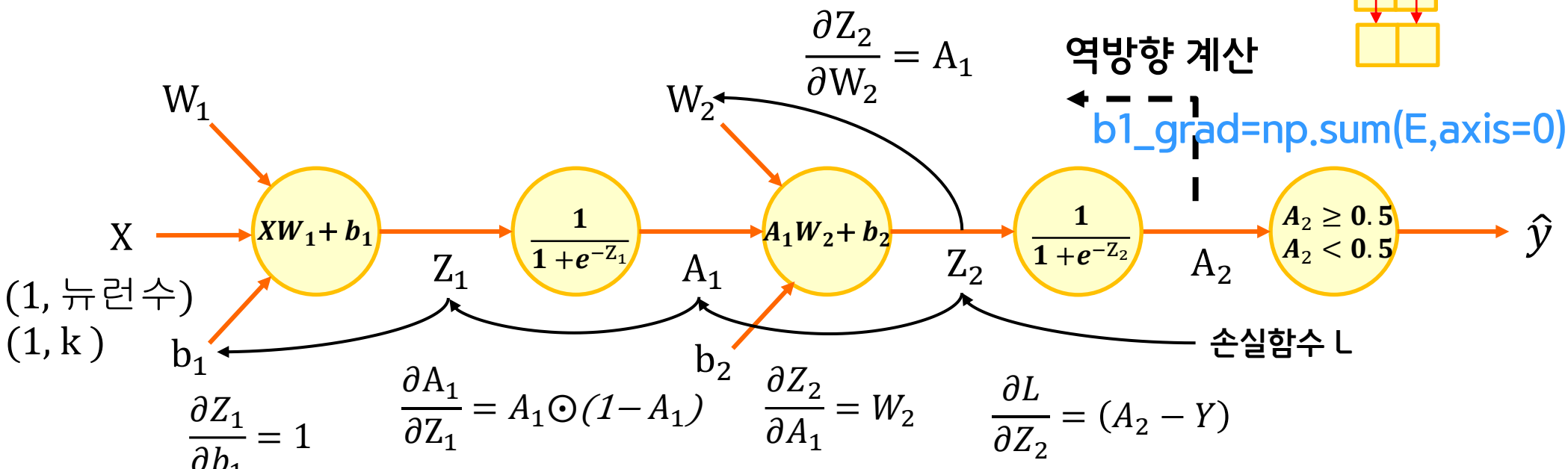
(m,1)

(1,m)(m, k) => (1,k)



역방향 계산

b1_grad=np.sum(E,axis=0)



SingleLayer 클래스를 상속한 DualLayer 클래스 구현

```
class DualLayer(SingleLayer):

    def __init__(self, units=10, learning_rate=0.1, l1=0, l2=0):
        self.units = units          # 은닉층의 뉴런 개수
        self.w1 = None              # 은닉층의 가중치
        self.b1 = None              # 은닉층의 절편
        self.w2 = None              # 출력층의 가중치
        self.b2 = None              # 출력층의 절편
        self.a1 = None              # 은닉층의 활성화 출력
        self.losses = []            # 훈련 손실
        self.val_losses = []        # 검증 손실
        self.lr = learning_rate     # 학습률
        self.l1 = l1                # L1 손실 하이퍼파라미터
        self.l2 = l2                # L2 손실 하이퍼파라미터
```

DualLayer는 은닉층과 출력층의 가중치와 절편을 각각 w1,b1 과 w2,b2에 저장한다.
은닉층의 활성화 출력은 역방향을 계산할 때 필요하므로 a1변수에 저장한다.

forpass() 함수 수정

```
def forpass(self, x):
        z1 = np.dot(x, self.w1) + self.b1
                                    # 첫 번째 층의 선형 식을 계산합니다
        self.a1 = self.activation(z1)
                                    # 활성화 함수를 적용합니다
        z2 = np.dot(self.a1, self.w2) + self.b2
                                    # 두 번째 층의 선형 식을 계산합니다.
        return z2
```

은닉층의 활성화 함수를 통과한 a1과 출력층의 가중치 w2를 곱하고 b2를 더하여 최종 출력 z2를 반환한다.
활정화 함수를 계산하는 activation() 함수는 SingleLayer 클래스로 부터 상속 받는다.

backprop() 함수 수정

```
def backprop(self, x, err):
        m = len(x)        # 샘플 개수
        # 출력층의 가중치와 절편에 대한 그래디언트를 계산합니다.
        w2_grad = np.dot(self.a1.T, err) / m
        b2_grad = np.sum(err) / m
        # 시그모이드 함수까지 그래디언트를 계산합니다.
        err_to_hidden = np.dot(err, self.w2.T) * self.a1 * (1 - self.a1)
        # 은닉층의 가중치와 절편에 대한 그래디언트를 계산합니다.
        w1_grad = np.dot(x.T, err_to_hidden) / m
        b1_grad = np.sum(err_to_hidden, axis=0) / m
        return w1_grad, b1_grad, w2_grad, b2_grad
```

$$\frac{\partial L}{\partial W_2} = \frac{\partial L}{\partial Z_2}\frac{\partial Z_2}{\partial W_2} = A_1^T(\underbrace{-(Y - A_2)}_{\text{err}}))$$

```
w2_grad = np.dot(self.a1.T, err) / m
```

$$\frac{\partial L}{\partial b_2} = \frac{\partial L}{\partial Z_2}\frac{\partial Z_2}{\partial b_2} = 1^{\mathrm{T}}(\underbrace{-(Y - A_2)}_{\text{err}}))$$

```
b2_grad = np.sum(err) / m
```

backprop() 함수 수정

$$X^T((A_2 - Y)\, W_2^T \odot A_1 \odot (1 - A_1))$$

```
def backprop(self, x, err):
        m = len(x)        # 샘플 개수
        # 출력층의 가중치와 절편에 대한 그래디언트를 계산합니다.
        w2_grad = np.dot(self.a1.T, err) / m
        b2_grad = np.sum(err) / m
        # 시그모이드 함수까지 그래디언트를 계산합니다.
        err_to_hidden = np.dot(err, self.w2.T) * self.a1 * (1 - self.a1)
        # 은닉층의 가중치와 절편에 대한 그래디언트를 계산합니다.
        w1_grad = np.dot(x.T, err_to_hidden) / m
        b1_grad = np.sum(err_to_hidden, axis=0) / m
        return w1_grad, b1_grad, w2_grad, b2_grad
```

$$\frac{\partial L}{\partial W_1} = \frac{\partial L}{\partial Z_2}\frac{\partial Z_2}{\partial A_1}\frac{\partial A_1}{\partial Z_1}\frac{\partial Z_1}{\partial W_1} = X^T \underbrace{((A_2 - Y)\, W_2^T \odot A_1 \odot (1 - A_1))}_{\texttt{err\_to\_hidden}}$$

```
err_to_hidden = np.dot(err, self.w2.T) * self.a1 * (1 - self.a1)
w1_grad = np.dot(x.T, err_to_hidden) / m      # 은닉층의 가중치에 대한 그래디언트 계산
b1_grad = np.sum(err_to_hidden, axis=0) / m   # 은닉층의 절편에 대한 그래디언트 계산
```

fit() 함수 수정 – 3개의 함수로 분리

```
def init_weights(self, n_features):
        self.w1 = np.ones((n_features, self.units))  # (특성 개수, 은닉층의 뉴런수)
        self.b1 = np.zeros(self.units)               # 은닉층의 뉴런수
        self.w2 = np.ones((self.units, 1))           # (은닉층의 뉴런수, 1)
        self.b2 = 0
```

```
def fit(self, x, y, epochs=100, x_val=None, y_val=None):
        y = y.reshape(-1, 1)          # 타깃을 열 벡터로 바꿉니다.
        y_val = y_val.reshape(-1, 1)
        m = len(x)                    # 샘플 개수를 저장합니다.
        self.init_weights(x.shape[1]) # 은닉층과 출력층의 가중치를 초기화합니다.
        # epochs만큼 반복합니다.
        for i in range(epochs):
            a = self.training(x, y, m)
            # 안전한 로그 계산을 위해 클리핑합니다.
            a = np.clip(a, 1e-10, 1-1e-10)
            # 로그 손실과 규제 손실을 더하여 리스트에 추가합니다.
            loss = np.sum(-(y*np.log(a) + (1-y)*np.log(1-a)))
            self.losses.append((loss + self.reg_loss()) / m)
            # 검증 세트에 대한 손실을 계산합니다.
            self.update_val_loss(x_val, y_val)
```

fit() 함수 수정 – 3개의 함수로 분리

```
def training(self, x, y, m):
        z = self.forpass(x)       # 정방향 계산을 수행합니다.
        a = self.activation(z)    # 활성화 함수를 적용합니다.
        err = a - y             # 오차를 계산합니다.
        # 오차를 역전파하여 그래디언트를 계산합니다.
        w1_grad, b1_grad, w2_grad, b2_grad = self.backprop(x, err)
        # 그래디언트에서 페널티 항의 미분 값을 뺍니다
        w1_grad += (self.l1 * np.sign(self.w1) + self.l2 * self.w1) / m
        w2_grad += (self.l1 * np.sign(self.w2) + self.l2 * self.w2) / m
        # 은닉층의 가중치와 절편을 업데이트합니다.
        self.w1 -= self.lr * w1_grad
        self.b1 -= self.lr * b1_grad
        # 출력층의 가중치와 절편을 업데이트합니다.
        self.w2 -= self.lr * w2_grad
        self.b2 -= self.lr * b2_grad
        return a
```

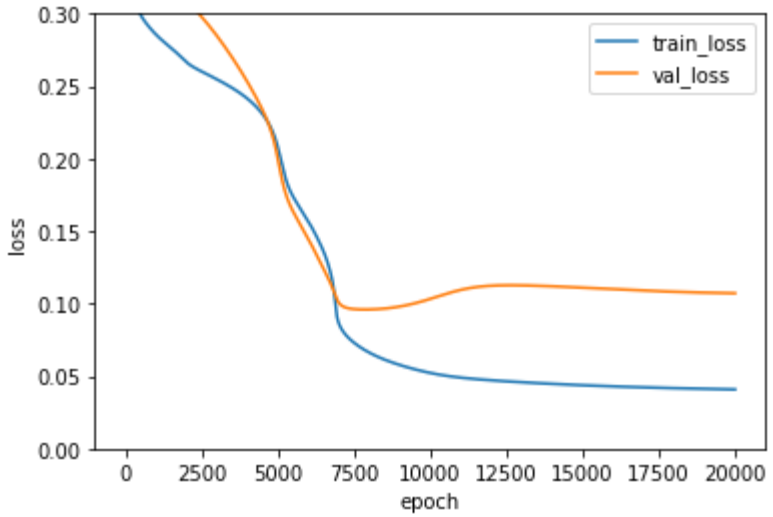
```
def reg_loss(self):
        # 은닉층과 출력층의 가중치에 규제를 적용합니다.
        return self.l1 * (np.sum(np.abs(self.w1)) + np.sum(np.abs(self.w2))) + \
               self.l2 / 2 * (np.sum(self.w1**2) + np.sum(self.w2**2))
```

다층 신경망 모델 훈련 및 평가

```
dual_layer = DualLayer(l2=0.01)
dual_layer.fit(x_train_scaled, y_train,
               x_val=x_val_scaled, y_val=y_val, epochs=20000)
dual_layer.score(x_val_scaled, y_val)
```

0.978021978021978

```
plt.ylim(0, 0.3)
plt.plot(dual_layer.losses)
plt.plot(dual_layer.val_losses)
plt.ylabel('loss')
plt.xlabel('epoch')
plt.legend(['train_loss', 'val_loss'])
plt.show()
```

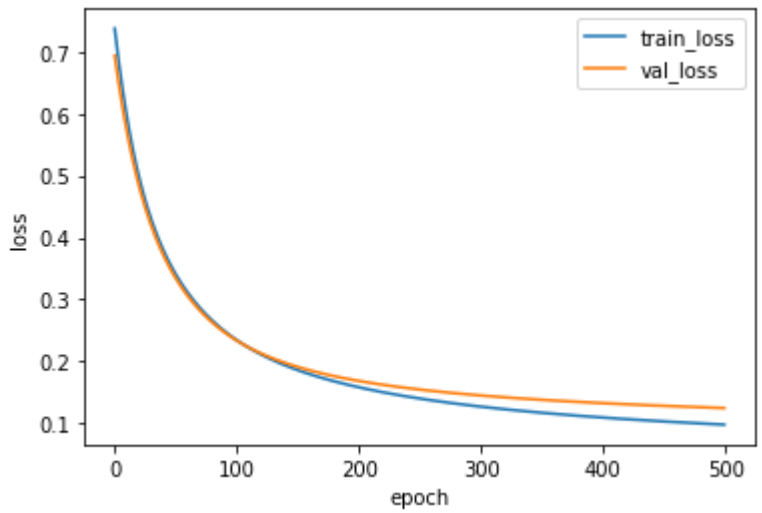


가중치 초기화 개선

```
class RandomInitNetwork(DualLayer):

    def init_weights(self, n_features):
        np.random.seed(42)
        self.w1 = np.random.normal(0, 1,
                                   (n_features, self.units))  # (특성 개수, 은닉층의 크기)
        self.b1 = np.zeros(self.units)                        # 은닉층의 크기
        self.w2 = np.random.normal(0, 1,
                                   (self.units, 1))           # (은닉층의 크기, 1)
        self.b2 = 0
```

```
random_init_net = RandomInitNetwork(l2=0.01)
random_init_net.fit(x_train_scaled, y_train,
                    x_val=x_val_scaled, y_val=y_val, epochs=500)
```

```
plt.plot(random_init_net.losses)
plt.plot(random_init_net.val_losses)
plt.ylabel('loss')
plt.xlabel('epoch')
plt.legend(['train_loss', 'val_loss'])
plt.show()
```



4. 다층 신경망 이해

4.1 행렬 곱 연산

4.2 배치경사 하강법 구현

4.3 2개의 층을 가진 신경망 구현

4.4 다층 신경망의 경사 하강법 구현

4.5 미니배치 경사 하강법 구현

4.6 다중분류 다층 신경망을 이해한다.

4.7 케라스를 이용한 다층신경망의 다양한 구현

미니배치 경사 하강법 이란?

배치 경사 하강법과 비슷하지만 에포크마다 전체 데이터를 사용하는 것이 아니라 조금씩 나누어 계산을 수행하고 그레디언트를 구하여 가중치를 업데이트 한다.

가중치 업데이트 방법

- 작게 나눈 미니 배치만큼 가중치를 업데이트 한다.
- 미니 배치의 크기는 보통 16,32,64등 2의 배수를 사용한다.
- 미니 배치의 크기가 1이면 확률적 경사 하강법이 된다.
- 미니 배치의 크기가 작으면 확률적 경사 하강법 처럼 작동하고 크면 배치 경사 하강법 처럼 작동한다.
- 미니 배치의 크기도 하이퍼파라미터이고 튜닝의 대상이다.

미니배치 경사 하강법 구현

__init__ () 함수에서 batch_size를 인자로 받아서 멤버 함수에 저장한다.

```
def __init__(self, units=10, batch_size=32, learning_rate=0.1, l1=0, l2=0):
        super().__init__(units, learning_rate, l1, l2)
        self.batch_size = batch_size     # 배치 크기
```

fit() 함수 수정

```
def fit(self, x, y, epochs=100, x_val=None, y_val=None):
        y_val = y_val.reshape(-1, 1)     # 타깃을 열 벡터로 바꿉니다.
        self.init_weights(x.shape[1])    # 은닉층과 출력층의 가중치를 초기화합니다.
        np.random.seed(42)
        # epochs만큼 반복합니다.
        for i in range(epochs):
            loss = 0
            # 제너레이터 함수에서 반환한 미니배치를 순환합니다.
            for x_batch, y_batch in self.gen_batch(x, y):
                y_batch = y_batch.reshape(-1, 1) # 타깃을 열 벡터로 바꿉니다.
                m = len(x_batch)                 # 샘플 개수를 저장합니다.
                a = self.training(x_batch, y_batch, m)
                # 안전한 로그 계산을 위해 클리핑합니다.
                a = np.clip(a, 1e-10, 1-1e-10)
                # 로그 손실과 규제 손실을 더하여 리스트에 추가합니다.
                loss += np.sum(-(y_batch*np.log(a) + (1-y_batch)*np.log(1-a)))
            self.losses.append((loss + self.reg_loss()) / len(x))
            # 검증 세트에 대한 손실을 계산합니다.
            self.update_val_loss(x_val, y_val)
```

self.gen_batch(x, y) 함수를 이용하여 특정한 range를 추출 하여
training 하여 w, b를 미니배치 마다 업데이트 시킨다.

364 // 32 => 몫 => 11+1 => 12

gen_batch() 함수 구현

```
# 미니배치 제너레이터 함수
    def gen_batch(self, x, y):
        length = len(x)
        bins = length // self.batch_size # 미니배치 횟수
        if length % self.batch_size:     # 32로 나누어 떨어지지 않는 나머지가 있는가?
            bins += 1                    # 나누어 떨어지지 않을 때
        indexes = np.random.permutation(np.arange(len(x))) # 인덱스를 섞습니다.
        x = x[indexes]
        y = y[indexes]
        for i in range(bins):
            start = self.batch_size * i
            end = self.batch_size * (i + 1)
            yield x[start:end], y[start:end]   # batch_size만큼 슬라이싱하여 반환합니다.
```

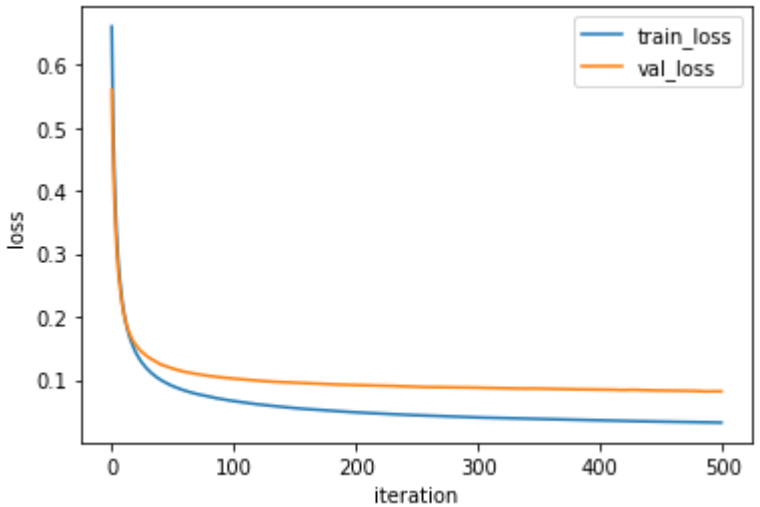
gen_batch() 함수는 파이썬 제너레이터로 구현한다.
제너레이터를 사용하면 명시적으로 리스트를 만들지 않으면서 필요한 만큼 데이터를 추출할 수 있으므로 효율적이다.
제너레이터 함수를 사용하려면 yield문을 사용하면 된다.
gen_batch() 함수는 batch_size 만큼씩 x,y 배열을 건너뛰며 미니 배치를 반환한다.
get_batch() 함수가 호출될 때마다 훈련 데이터 배열의 인덱스를 섞는다.

미니배치 하강법 적용 (batch_size = 32)

```
minibatch_net = MinibatchNetwork(l2=0.01, batch_size=32)
minibatch_net.fit(x_train_scaled, y_train,
                  x_val=x_val_scaled, y_val=y_val, epochs=500)
minibatch_net.score(x_val_scaled, y_val)
```

0.978021978021978

```
plt.plot(minibatch_net.losses)
plt.plot(minibatch_net.val_losses)
plt.ylabel('loss')
plt.xlabel('iteration')
plt.legend(['train_loss', 'val_loss'])
plt.show()
```

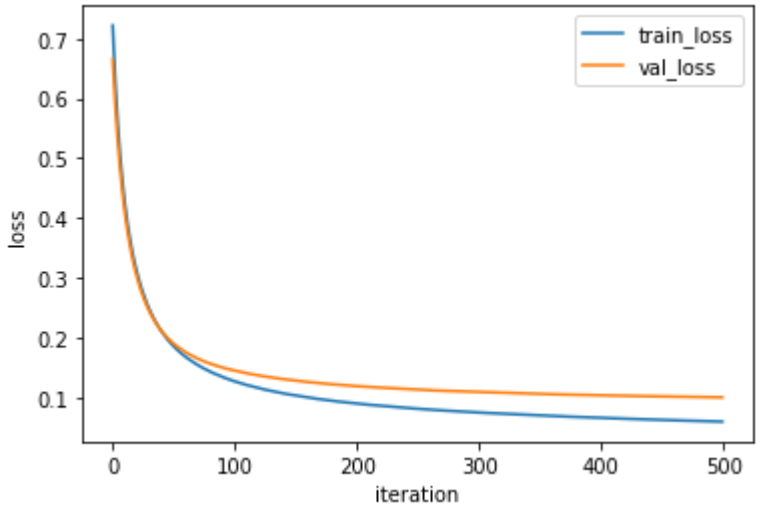


미니배치 하강법 적용 (batch_size = 128)

```
minibatch_net = MinibatchNetwork(l2=0.01, batch_size=128)
minibatch_net.fit(x_train_scaled, y_train,
                  x_val=x_val_scaled, y_val=y_val, epochs=500)
minibatch_net.score(x_val_scaled, y_val)
```

0.978021978021978

```
plt.plot(minibatch_net.losses)
plt.plot(minibatch_net.val_losses)
plt.ylabel('loss')
plt.xlabel('iteration')
plt.legend(['train_loss', 'val_loss'])
plt.show()
```



4. 다층 신경망 이해

4.1 행렬 곱 연산

4.2 배치경사 하강법 구현

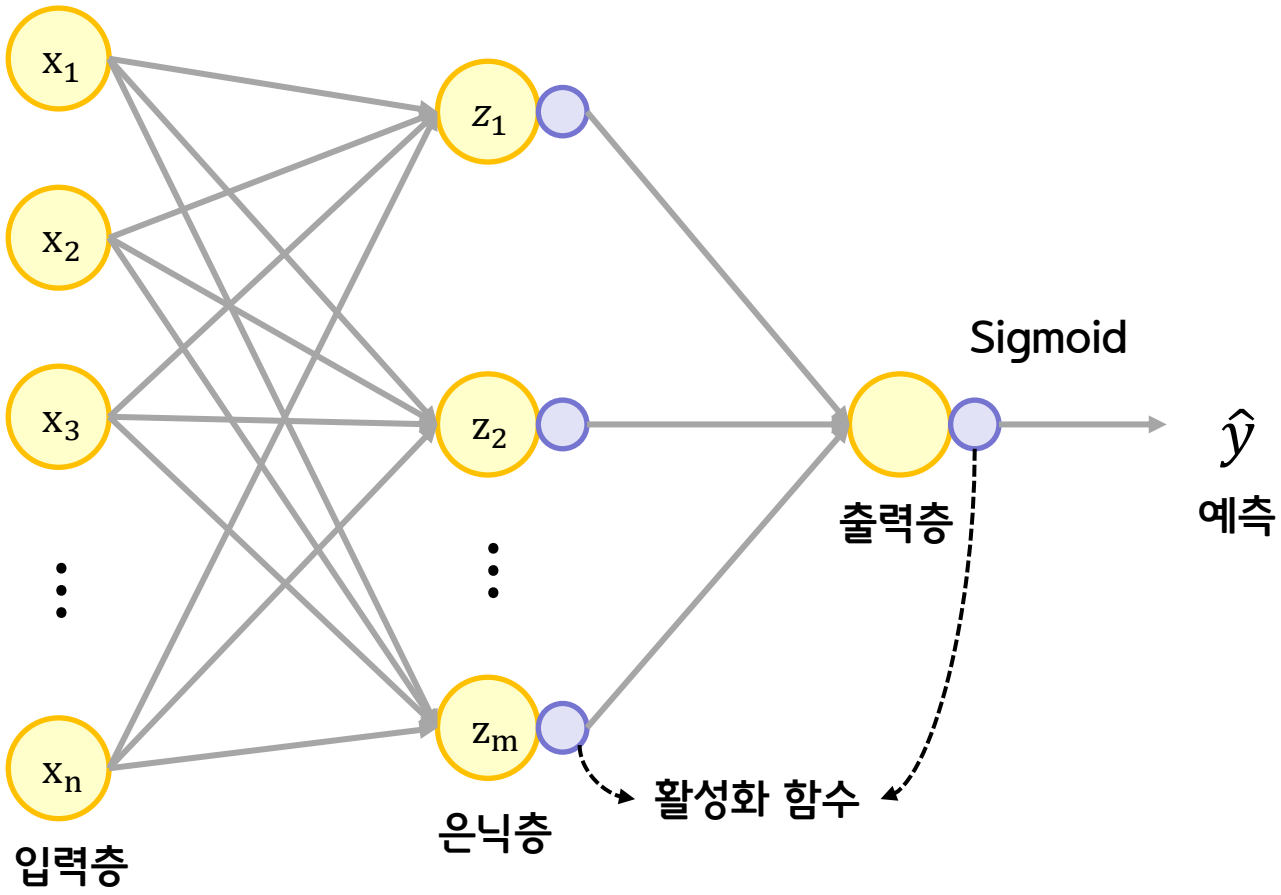
4.3 2개의 층을 가진 신경망 구현

4.4 다층 신경망의 경사 하강법 구현

4.5 미니배치 경사 하강법 구현

4.6 다중분류 다층 신경망을 이해한다.

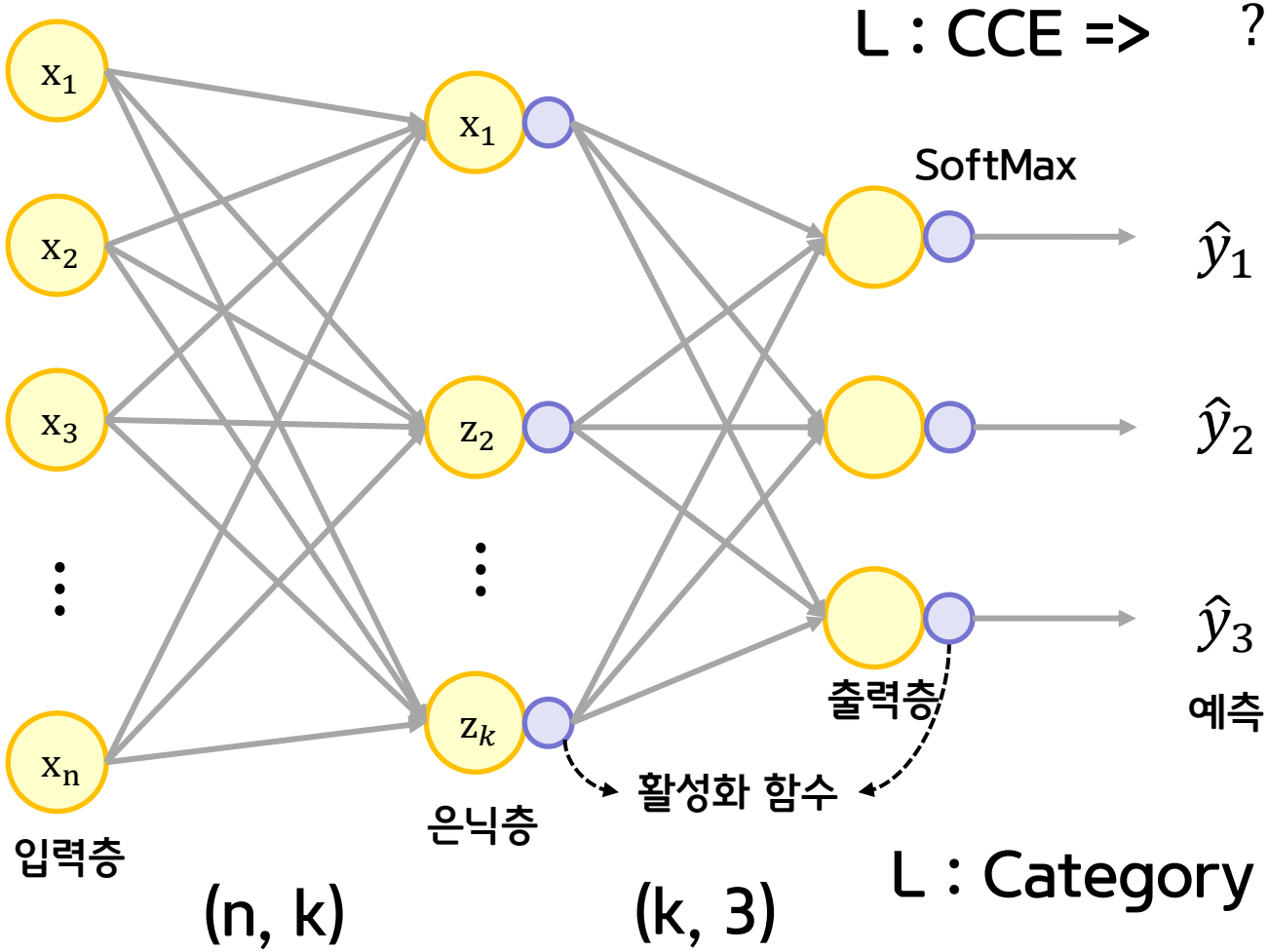
4.7 케라스를 이용한 다층신경망의 다양한 구현


x_1
x_2
x_3
x_n
입력층
z_1
z_2
z_m
은닉층
Sigmoid
출력층
$\hat{y}$
예측
활성화 함수

L : MSE => $(\hat{y} - y)$

L : BCE => $(a - y)$

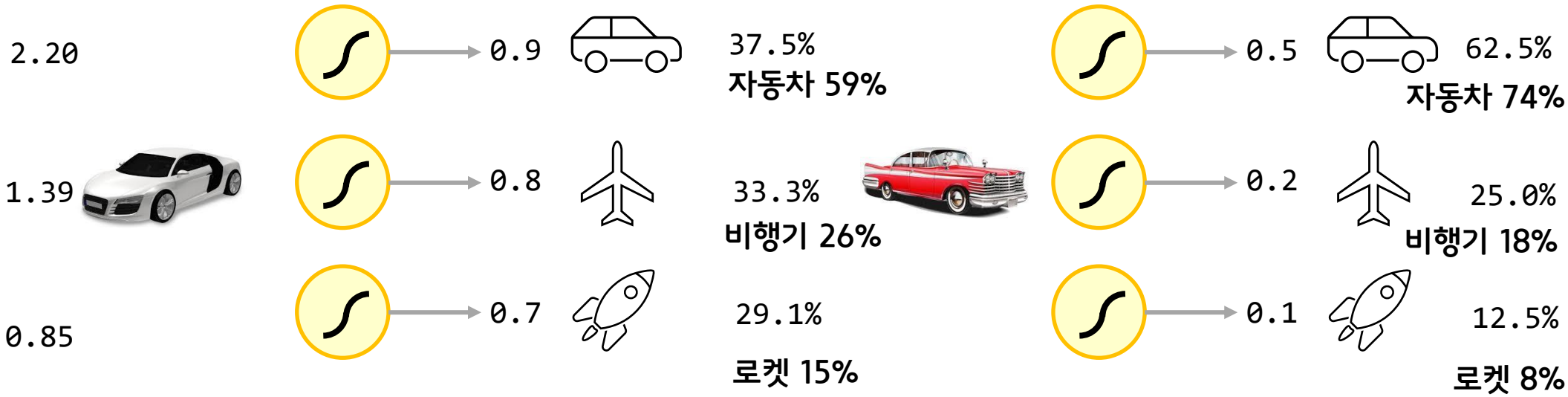
L : CCE => ??



L : Category Cross Entropy

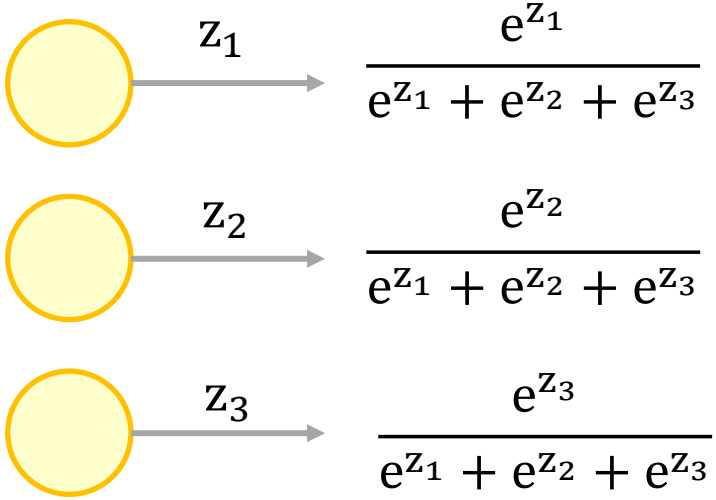
활성화 출력의 합이 1이 아니면 비교하기 어렵다.

소프트맥스 함수를 적용해 출력 강도를 정규화 한다.



소프트맥스 함수를 적용해 출력 강도를 정규화 한다.

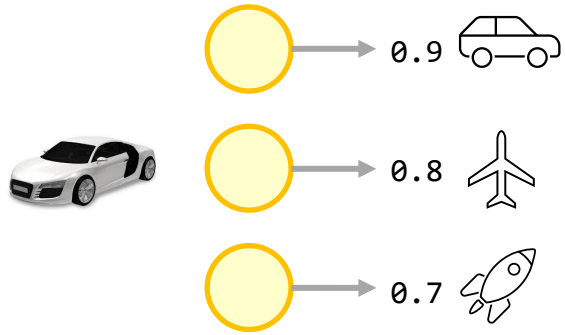
$$\frac{e^{z_i}}{e^{z_1} + e^{z_2} + e^{z_3}}$$



소프트맥스 함수를 적용해 출력 강도를 정규화 한다.

시그 모이드 함수를 z에 대해 정리하면

$$\hat{y} = \frac{1}{1 + e^{-z}} \Rightarrow z = -\log\left(\frac{1}{\hat{y}} - 1\right)$$



$$z_1 = -\log\left(\frac{1}{0.9} - 1\right) = 2.20 \qquad z_2 = -\log\left(\frac{1}{0.8} - 1\right) = 1.39 \qquad z_2 = -\log\left(\frac{1}{0.7} - 1\right) = 0.85$$

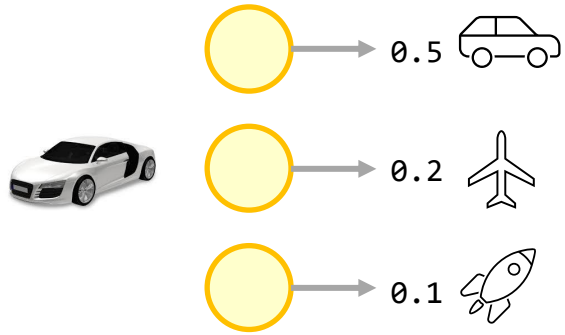
$$\hat{y} = \frac{e^{2.20}}{e^{2.20}+e^{1.39}+e^{0.85}} = 0.59 \qquad \hat{y} = \frac{e^{1.39}}{e^{2.20}+e^{1.39}+e^{0.85}} = 0.26 \qquad \hat{y} = \frac{e^{0.85}}{e^{2.20}+e^{1.39}+e^{0.85}} = 0.15$$

자동차 59% 비행기 26% 로켓 15%

소프트맥스 함수를 적용해 출력 강도를 정규화 한다.

시그 모이드 함수를 z에 대해 정리하면

$$\hat{y} = \frac{1}{1 + e^{-z}} \Rightarrow z = -\log\left(\frac{1}{\hat{y}} - 1\right)$$



$$z_1 = -\log\left(\frac{1}{0.5} - 1\right) = 0.00 \qquad z_2 = -\log\left(\frac{1}{0.2} - 1\right) = -1.39 \qquad z_2 = -\log\left(\frac{1}{0.1} - 1\right) = -2.20$$

$$\hat{y} = \frac{e^{0.00}}{e^{0.00}+e^{-1.39}+e^{-2.20}} = 0.74 \qquad \hat{y} = \frac{e^{-1.39}}{e^{0.00}+e^{-1.39}+e^{-2.20}} = 0.18 \qquad \hat{y} = \frac{e^{-2.20}}{e^{0.00}+e^{-1.39}+e^{-2.20}} = 0.08$$

자동차 74% 비행기 18% 로켓 8%

크로스 엔트로피 손실 함수

$$L = -\sum_{c=1}^{10} y_c \log(a_c) = -(y_1 \log(a_1) + y_2 \log(a_2) + \cdots + y_c \log(a_c))$$

크로스 엔트로피 손실 함수 미분

$$\frac{\partial L}{\partial z_1} = \frac{\partial L}{\partial a_1}\frac{\partial a_1}{\partial z_1} + \frac{\partial L}{\partial a_2}\frac{\partial a_2}{\partial z_1} + \frac{\partial L}{\partial a_3}\frac{\partial a_3}{\partial z_1}$$

크로스 엔트로피 손실 함수 미분

$$\frac{\partial L}{\partial z_1} = \frac{\partial L}{\partial a_1}\frac{\partial a_1}{\partial z_1} + \frac{\partial L}{\partial a_2}\frac{\partial a_2}{\partial z_1} + \frac{\partial L}{\partial a_3}\frac{\partial a_3}{\partial z_1}$$

$$\frac{\partial L}{\partial a_1} = -\frac{y_1}{a_1} \qquad \frac{\partial L}{\partial a_2} = -\frac{y_2}{a_2} \qquad \frac{\partial L}{\partial a_3} = -\frac{y_3}{a_3}$$

$$\frac{\partial L}{\partial z_1} = \left(-\frac{y_1}{a_1}\right)\frac{\partial a_1}{\partial z_1} + \left(-\frac{y_2}{a_2}\right)\frac{\partial a_2}{\partial z_1} + \left(-\frac{y_3}{a_3}\right)\frac{\partial a_3}{\partial z_1}$$

크로스 엔트로피 손실 함수 미분

$$\frac{\partial \mathrm{L}}{\partial \mathrm{z}_1} = \left(-\frac{\mathrm{y}_1}{\mathrm{a}_1}\right)\frac{\partial \mathrm{a}_1}{\partial \mathrm{z}_1} + \left(-\frac{\mathrm{y}_2}{\mathrm{a}_2}\right)\frac{\partial \mathrm{a}_2}{\partial \mathrm{z}_1} + \left(-\frac{\mathrm{y}_3}{\mathrm{a}_3}\right)\frac{\partial \mathrm{a}_3}{\partial \mathrm{z}_1}$$

$$\frac{\partial \mathrm{a}_1}{\partial \mathrm{z}_1} = \mathrm{a}_1(1-\mathrm{a}_1) \qquad \frac{\partial \mathrm{a}_2}{\partial \mathrm{z}_1} = -\mathrm{a}_2\mathrm{a}_1 \qquad \frac{\partial \mathrm{a}_3}{\partial \mathrm{z}_1} = -\mathrm{a}_3\mathrm{a}_1$$

$$\frac{\partial \mathrm{L}}{\partial \mathrm{z}_1} = \left(-\frac{\mathrm{y}_1}{\mathrm{a}_1}\right)\mathrm{a}_1(1-\mathrm{a}_1) + \left(-\frac{\mathrm{y}_2}{\mathrm{a}_2}\right)(-\mathrm{a}_2\mathrm{a}_1) + \left(-\frac{\mathrm{y}_3}{\mathrm{a}_3}\right)(-\mathrm{a}_3\mathrm{a}_1)$$
$$= (\mathrm{a}_1 - y_1)$$

$$\frac{\boldsymbol{\partial L}}{\boldsymbol{\partial z}} = (\boldsymbol{a} - \boldsymbol{y})$$

다중 분류 신경망 구현

미니배치 하강법 적용 (batch_size = 128)

```
def sigmoid(self, z):
    z = np.clip(z, -100, None)            # 안전한 np.exp() 계산을 위해
    a = 1 / (1 + np.exp(-z))              # 시그모이드 계산
    return a

def softmax(self, z):
    # 소프트맥스 함수
    z = np.clip(z, -100, None)            # 안전한 np.exp() 계산을 위해
    exp_z = np.exp(z)
    return exp_z / np.sum(exp_z, axis=1).reshape(-1, 1)
```

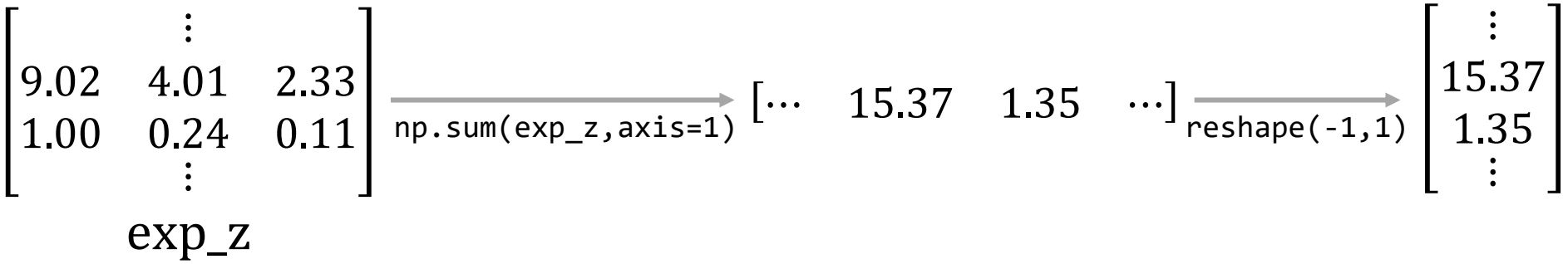
$$\begin{bmatrix} \vdots & \vdots & \vdots \\ 2.20 & 1.39 & 0.85 \\ 0.00 & -1.39 & -2.20 \\ \vdots & \vdots & \vdots \end{bmatrix} \xrightarrow{\texttt{np.exp(z)}} \begin{bmatrix} \vdots & \vdots & \vdots \\ e^{2.20} & e^{1.39} & e^{0.85} \\ e^{0.00} & e^{-1.39} & e^{-2.20} \\ \vdots & \vdots & \vdots \end{bmatrix}$$

z $\qquad$ e^z

다중 분류 신경망 구현

미니배치 하강법 적용 (batch_size = 128)

```
def sigmoid(self, z):
    z = np.clip(z, -100, None)            # 안전한 np.exp() 계산을 위해
    a = 1 / (1 + np.exp(-z))              # 시그모이드 계산
    return a

def softmax(self, z):
    # 소프트맥스 함수
    z = np.clip(z, -100, None)            # 안전한 np.exp() 계산을 위해
    exp_z = np.exp(z)
    return exp_z / np.sum(exp_z, axis=1).reshape(-1, 1)
```



다중 분류 신경망 구현

미니배치 하강법 적용 (batch_size = 128)

```
def sigmoid(self, z):
    z = np.clip(z, -100, None)            # 안전한 np.exp() 계산을 위해
    a = 1 / (1 + np.exp(-z))              # 시그모이드 계산
    return a

def softmax(self, z):
    # 소프트맥스 함수
    z = np.clip(z, -100, None)            # 안전한 np.exp() 계산을 위해
    exp_z = np.exp(z)
    return exp_z / np.sum(exp_z, axis=1).reshape(-1, 1)
```

$$\begin{bmatrix} \vdots & & \\ 9.02 & 4.01 & 2.33 \\ 1.00 & 0.24 & 0.11 \\ & \vdots & \end{bmatrix} \div \begin{bmatrix} \vdots \\ 15.37 \\ 1.35 \\ \vdots \end{bmatrix} = \begin{bmatrix} & \vdots & \\ 0.59 & 0.26 & 0.15 \\ 0.74 & 0.18 & 0.08 \\ & \vdots & \end{bmatrix}$$

exp_z

np.sum(exp_z,axis=1).reshape(-1,1)

다중 분류 신경망 구현

가중치 초기화

```
def init_weights(self, n_features, n_classes):
    self.w1 = np.random.normal(0, 1,
                               (n_features, self.units))  # (특성 개수, 은닉층의 크기)
    self.b1 = np.zeros(self.units)                        # 은닉층의 크기
    self.w2 = np.random.normal(0, 1,
                               (self.units, n_classes))   # (은닉층의 크기, 클래스 개수)
    self.b2 = np.zeros(n_classes)
```

다중 분류 신경망 구현

fit() 함수 수정

```
def fit(self, x, y, epochs=100, x_val=None, y_val=None):
    np.random.seed(42)
    self.init_weights(x.shape[1], y.shape[1])    # 은닉층과 출력층의 가중치를 초기화합니다.
    # epochs만큼 반복합니다.
    for i in range(epochs):
        loss = 0
        print('.', end='')
        # 제너레이터 함수에서 반환한 미니배치를 순환합니다.
        for x_batch, y_batch in self.gen_batch(x, y):
            a = self.training(x_batch, y_batch)
            # 안전한 로그 계산을 위해 클리핑합니다.
            a = np.clip(a, 1e-10, 1-1e-10)
            # 로그 손실과 규제 손실을 더하여 리스트에 추가합니다.
            loss += np.sum(-y_batch*np.log(a))
        self.losses.append((loss + self.reg_loss()) / len(x))
        # 검증 세트에 대한 손실을 계산합니다.
        self.update_val_loss(x_val, y_val)
```

다중 분류 신경망 구현

training() 함수 수정

```
def training(self, x, y):
    m = len(x)                  # 샘플 개수를 저장합니다.
    z = self.forpass(x)         # 정방향 계산을 수행합니다.
    a = self.softmax(z)         # 활성화 함수를 적용합니다.
```

predict() 함수 수정

```
def predict(self, x):
    z = self.forpass(x)             # 정방향 계산을 수행합니다.
    return np.argmax(z, axis=1)     # 가장 큰 값의 인덱스를 반환합니다.
```

score() 함수 수정

```
def score(self, x, y):
    # 예측과 타깃 열 벡터를 비교하여 True의 비율을 반환합니다.
    return np.mean(self.predict(x) == np.argmax(y, axis=1))
```

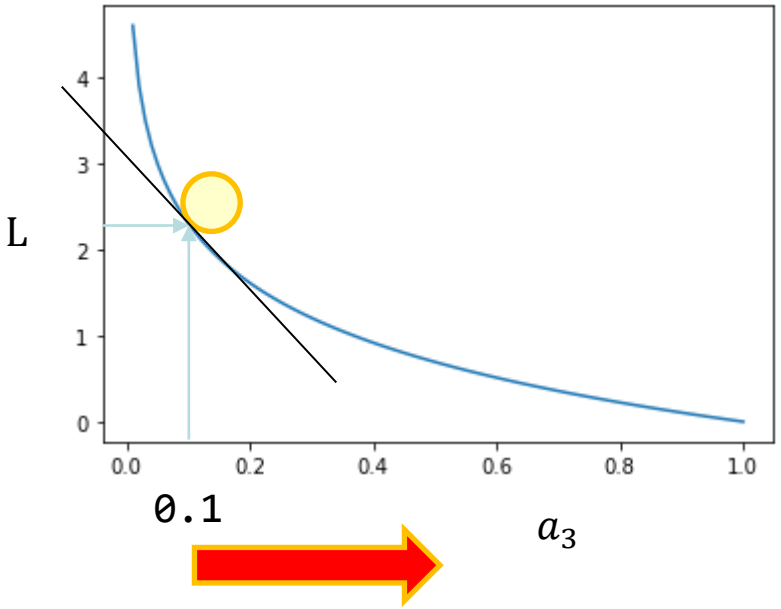
다중 분류 신경망 구현

검증 손실 계산

```
def update_val_loss(self, x_val, y_val):
    z = self.forpass(x_val)             # 정방향 계산을 수행합니다.
    a = self.softmax(z)                 # 활성화 함수를 적용합니다.
    a = np.clip(a, 1e-10, 1-1e-10)      # 출력 값을 클리핑합니다.
    # 크로스 엔트로피 손실과 규제 손실을 더하여 리스트에 추가합니다.
    val_loss = np.sum(-y_val*np.log(a))
    self.val_losses.append((val_loss + self.reg_loss()) / len(y_val))
```

```
          개     고양이     사슴
a = [0.7,   0.2,      0.1]
y = [  0,     0,        1]
```

$$L = -\sum_{c=1}^{3} y_c \log(a_c) = -(y_1 \log(a_1) + y_2 \log(a_2) + y_3 \log(a_3))$$

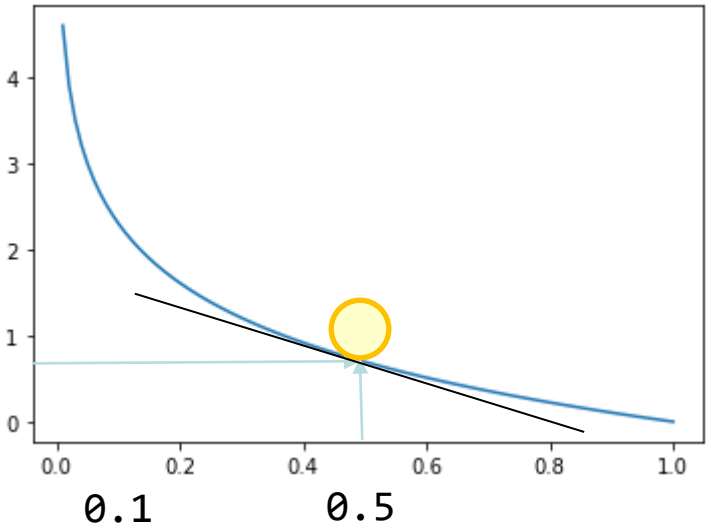



```
err = (a-y)
err = (0.1-1) = -0.9
```

개 고양이 사슴

```
a = [0.1,  0.4,    0.5]
y = [  0,    0,      1]
```

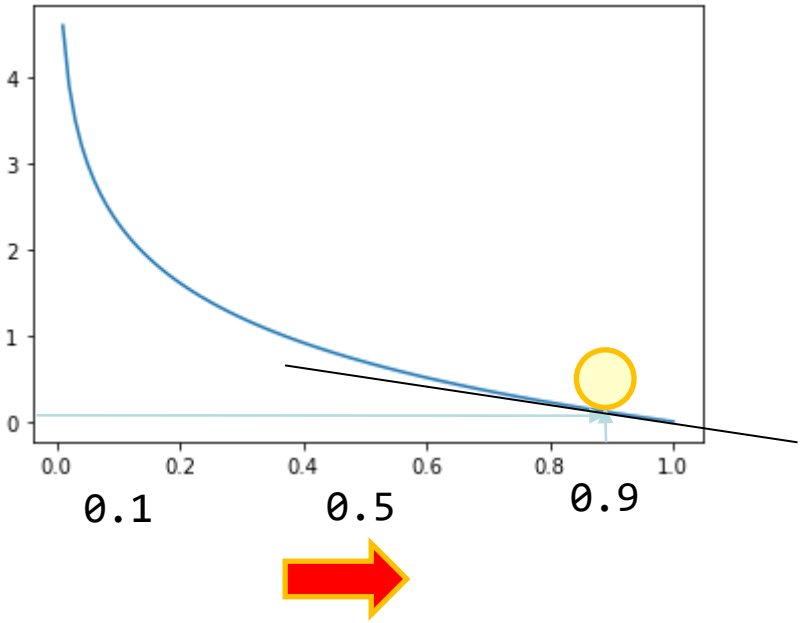
$$L = -\sum_{c=1}^{3} y_c \log(a_c) = -(y_1 \log(a_1) + y_2 \log(a_2) + y_3 \log(a_3))$$




```
err = (a-y)
err = (0.5-1) = -0.5
```

```
          개      고양이     사슴
a = [0.0,   0.1,      0.9]
y = [  0,     0,        1]
```

$$L = -\sum_{c=1}^{3} y_c \log(a_c) = -(y_1 \log(a_1) + y_2 \log(a_2) + y_3 \log(a_3))$$




```
err = (a-y)
err = (0.9-1) = -0.1
```

4. 다층 신경망 이해

4.1 행렬 곱 연산

4.2 배치경사 하강법 구현

4.3 2개의 층을 가진 신경망 구현

4.4 다층 신경망의 경사 하강법 구현

4.5 미니배치 경사 하강법 구현

4.6 다중분류 다층 신경망을 이해한다.

4.7 케라스를 이용한 다층신경망의 다양한 구현

텐서플로 임포트

```
import tensorflow as tf
```

텐서플로 버전 확인

```
tf.__version__
```

```
'2.1.0'
```

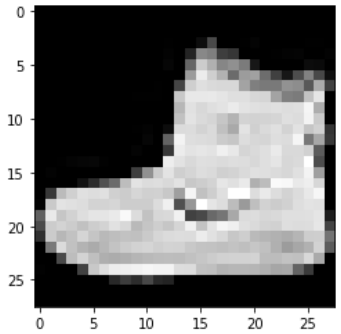
패션 MNIST 데이터 세트 불러오기

```
(x_train_all, y_train_all), (x_test, y_test) = tf.keras.datasets.fashion_mnist.load_data()
print(x_train_all.shape, y_train_all.shape)
```

```
(60000, 28, 28) (60000,)
```

패션 MNIST 데이터 세트 불러오기

```
import matplotlib.pyplot as plt

plt.imshow(x_train_all[0], cmap='gray')
plt.show()
```



타깃의 내용과 의미 확인

```
print(y_train_all[:10])
```

```
 [9 0 0 3 0 2 7 2 5 5]
```

```
class_names = ['티셔츠/윗도리', '바지', '스웨터', '드레스', '코트',
               '샌들', '셔츠', '스니커즈', '가방', '앵클부츠']
print(class_names[y_train_all[0]])
```

```
'앵클부츠'
```

타깃 분포 확인

```
np.bincount(y_train_all)
```

```
array([6000, 6000, 6000, 6000, 6000, 6000, 6000, 6000, 6000, 6000], dtype=int64)
```

훈련 세트와 검증 세트 고르게 나누기

```
from sklearn.model_selection import train_test_split
x_train, x_val, y_train, y_val = train_test_split(x_train_all, y_train_all,
stratify=y_train_all, test_size=0.2, random_state=42)
```

```
np.bincount(y_train)
```

```
array([4800, 4800, 4800, 4800, 4800, 4800, 4800, 4800, 4800, 4800], dtype=int64)
```

```
np.bincount(y_val)
```

```
array([1200, 1200, 1200, 1200, 1200, 1200, 1200, 1200, 1200, 1200], dtype=int64)
```

입력 데이터 정규화

```
x_train = x_train / 255
x_val = x_val / 255
```

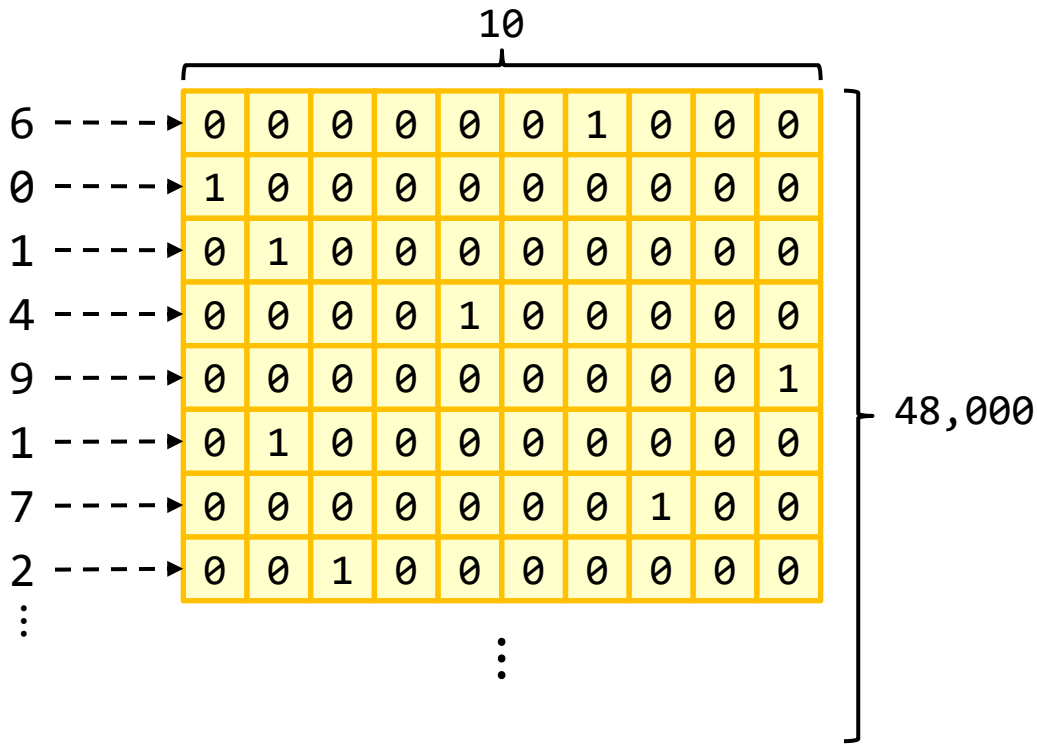
훈련 세트와 검증 세트의 차원 변경

```
x_train = x_train.reshape(-1, 784)
x_val = x_val.reshape(-1, 784)
print(x_train.shape, x_val.shape)
```

```
(48000, 784) (12000, 784)
```

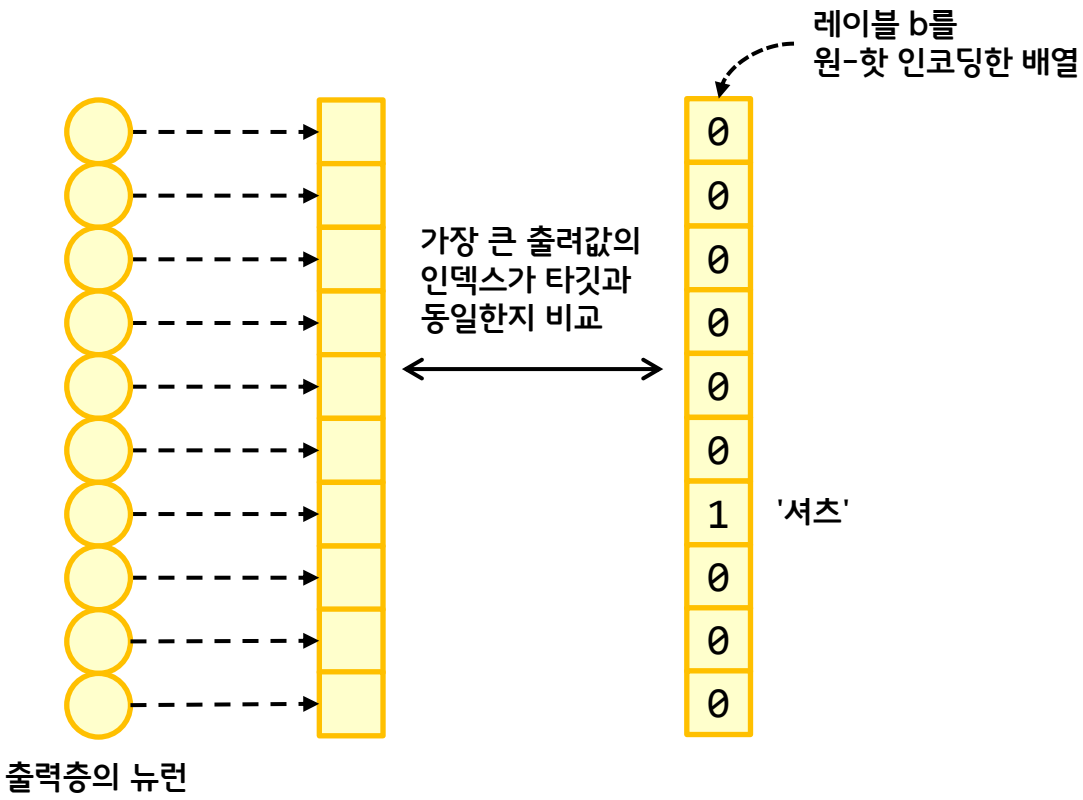
타깃 데이터를 준비하고 다중 분류 신경망을 훈련한다.

타깃을 원-핫 인코딩으로 변환



타깃 데이터를 준비하고 다중 분류 신경망을 훈련한다.

타깃을 원-핫 인코딩으로 변환



to_categorical() 함수 사용해 원-핫 인코딩하기

```
tf.keras.utils.to_categorical([0, 1, 3])
```

```
array([[1., 0., 0., 0.],
       [0., 1., 0., 0.],
       [0., 0., 0., 1.]], dtype=float32)
```

```
y_train_encoded = tf.keras.utils.to_categorical(y_train)
y_val_encoded = tf.keras.utils.to_categorical(y_val)
print(y_train_encoded.shape, y_val_encoded.shape)
```

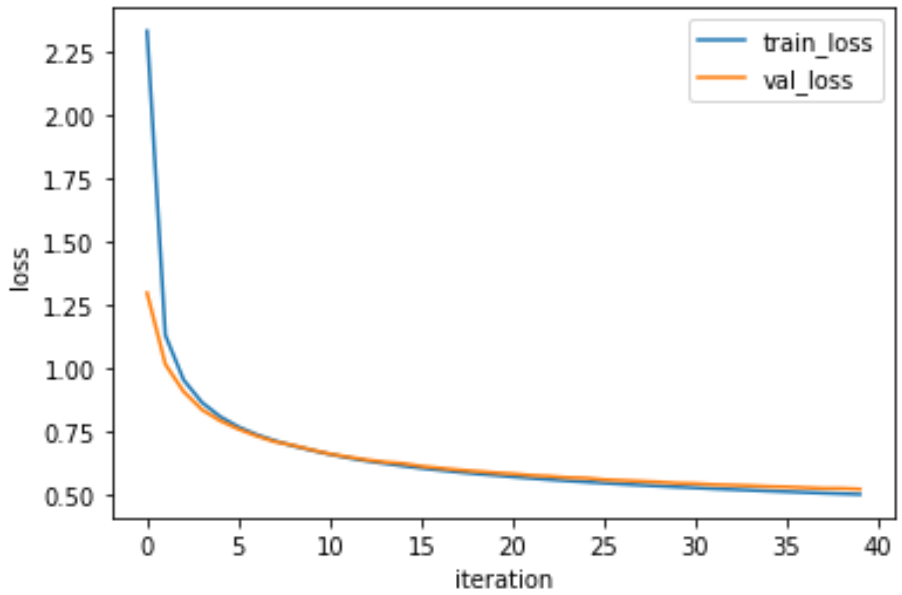
```
(48000, 10) (12000, 10)
```

```
print(y_train[0], y_train_encoded[0])
```

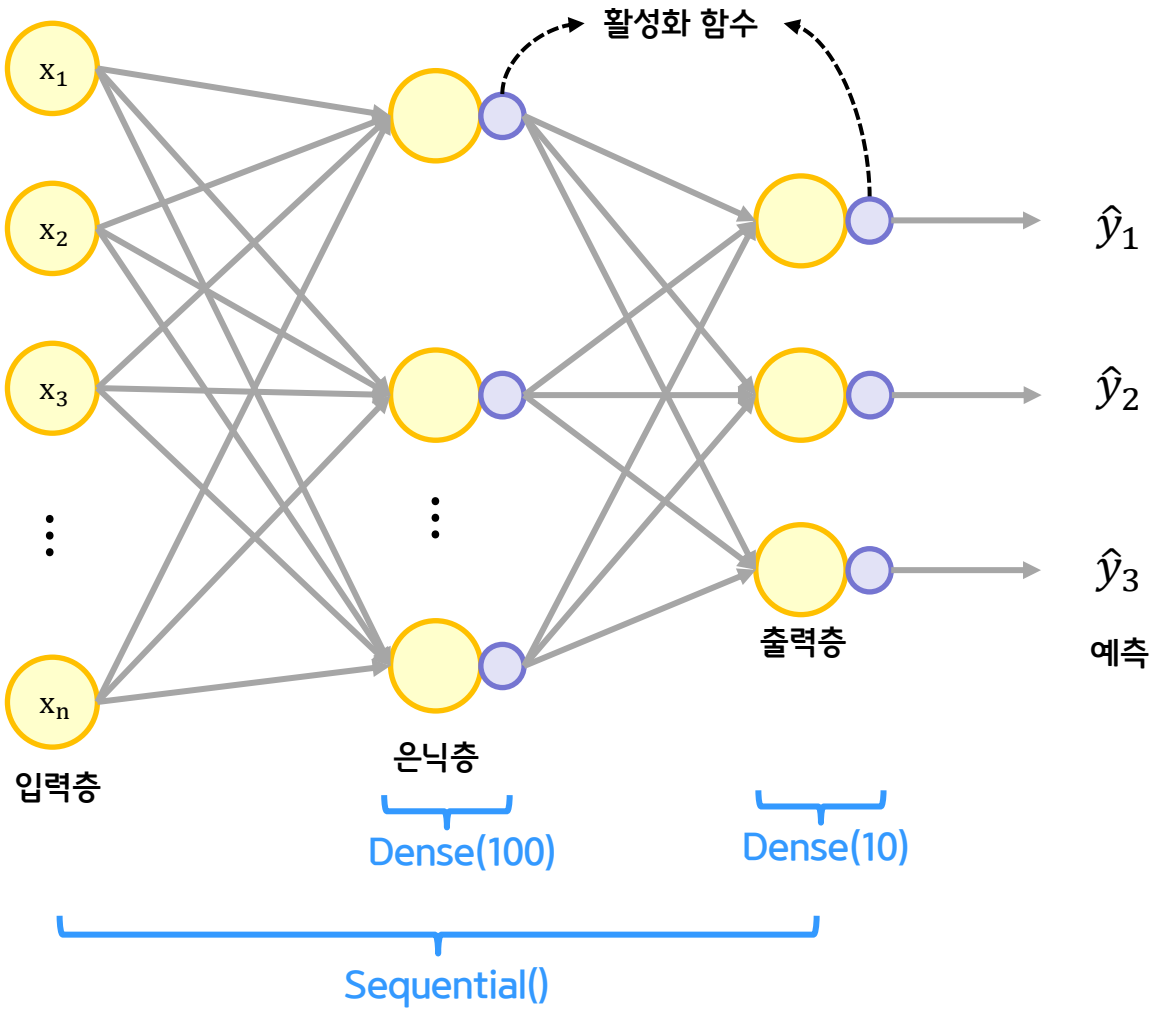
```
6 [0. 0. 0. 0. 0. 0. 1. 0. 0. 0.]
```

MultiClassNetwork 클래스로 다중 분류 신경망 훈련하기

```
fc = MultiClassNetwork(units=100, batch_size=256)
fc.fit(x_train, y_train_encoded,
       x_val=x_val, y_val=y_val_encoded, epochs=40)
plt.plot(fc.losses)
plt.plot(fc.val_losses)
plt.ylabel('loss')
plt.xlabel('iteration')
plt.legend(['train_loss', 'val_loss'])
plt.show()
```



케라스로 다중 분류 신경망 훈련하기



케라스로 다중 분류 신경망 훈련하기

모델 생성

```
from tensorflow.keras import Sequential
from tensorflow.keras.layers import Dense
model = Sequential()
```

은닉층과 출력층을 모델에 추가

```
model.add(Dense(100, activation='sigmoid', input_shape=(784,)))
model.add(Dense(10, activation='softmax'))
```

최적화 알고리즘과 손실 함수 지정

```
model.compile(optimizer='sgd', loss='categorical_crossentropy',
              metrics=['accuracy'])
```

모델 훈련하기

```
history = model.fit(x_train, y_train_encoded, epochs=40,
                    validation_data=(x_val, y_val_encoded))
```

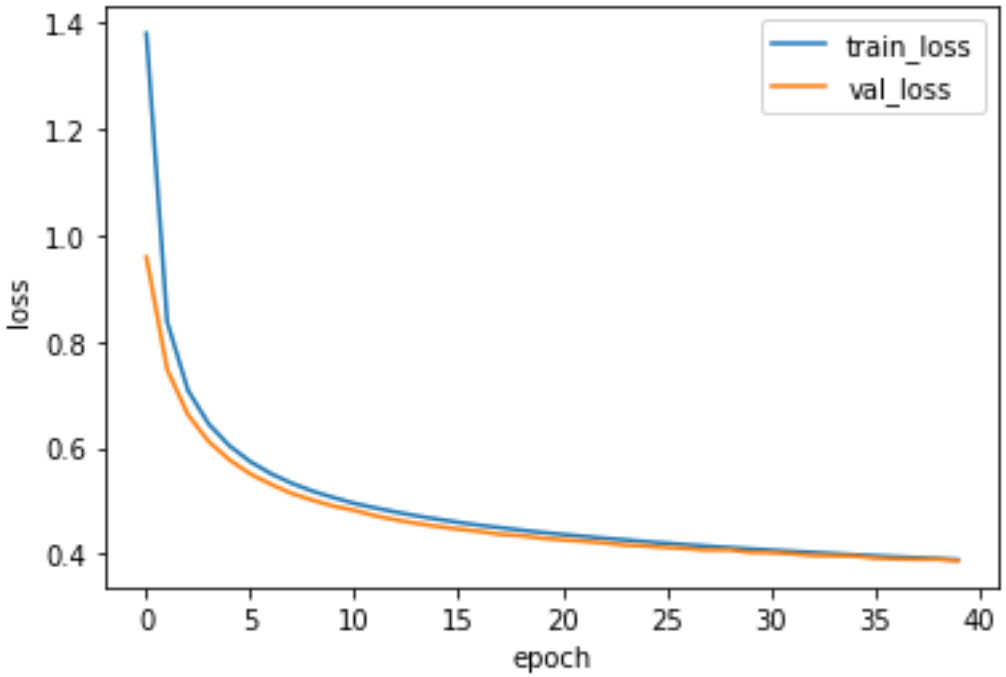
history객체의 history 딕셔너리

```
print(history.history.keys())
```

```
dict_keys(['loss', 'accuracy', 'val_loss', 'val_accuracy'])
```

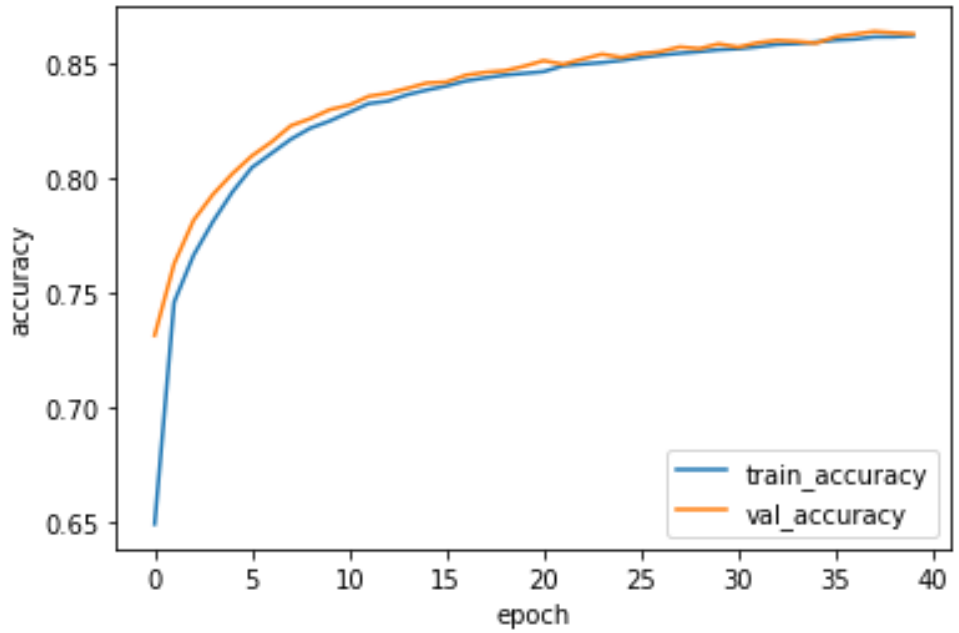
손실 그래프

```
plt.plot(history.history['loss'])
plt.plot(history.history['val_loss'])
plt.ylabel('loss')
plt.xlabel('epoch')
plt.legend(['train_loss', 'val_loss'])
plt.show()
```



정확도 그래프

```
plt.plot(history.history['accuracy'])
plt.plot(history.history['val_accuracy'])
plt.ylabel('accuracy')
plt.xlabel('epoch')
plt.legend(['train_accuracy', 'val_accuracy'])
plt.show()
```



검증 세트 정확도 계산

```
loss, accuracy = model.evaluate(x_val, y_val_encoded, verbose=0)
print(accuracy)
```

```
0.86291665
```